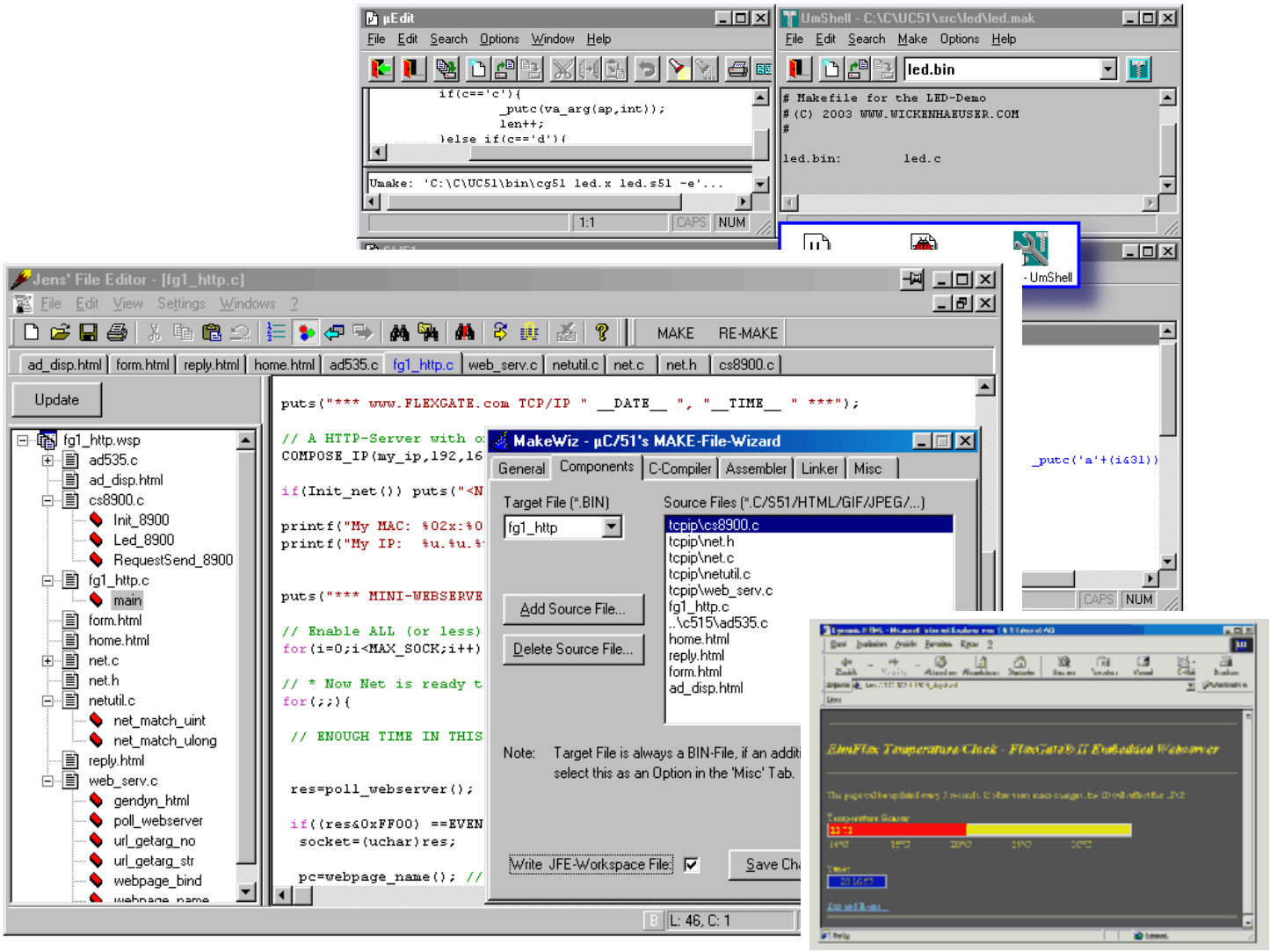
µC/51 V1.20.04
User's Manual



ANSI C development system for the 8051 family of microcontrollers with integral TCP/IP support

Preface 1
History 1
Target processors 1
Source code debugging with SiLabs CPUs (formerly Cygnal) 2
Code quality 2
TCP/IP Stack and Internet connecticity 2
Limitations of the demo version 4
Sourcecode compatibility with other compilers 4
Full version 4
Support - Versions 5
About us 5
Your support is wanted 5
Credits 5
The future 5
Disclaimer 5
Errors 6
Trademarks 6
Internet 6

Installation and de installation 6
Quick start - Desktop symbols 6
Obtaining a license for the full version - distributors. 7
Installing the full version 7
Updates - Installing new versions - Keeping a license 8
Transferring a license (New since V1.10.12) 8
Documentation 8

Quick start - a micro tutorial - The NEW V1.20 version! . . 8

Quick start - a micro tutorial - The old V1.10 version! . . . 10
Way 1.): Single chip microcontrollers - series production boards 11
Hardware requirements 11
First demo: SC_HELLO - a single chip "Hello World" 11
Second demo: KITCLOCK - a kitchen clock 12
MENUE/MENUE2 demo - a good starting point for own applications 12
A89S8252 demo - accessing the internal E2PROM 12
Notes about the MAKEFILES 12
Way 2.): Microcontroller hardware - special development boards 13
Hello World! 14

MakeWiz - a closer description 15
The 'General' Tab 16
The 'Components' Tab 16
The 'C-Compiler' Tab 16
The 'Assembler' Tab 16
The 'Linker' Tab 17
The 'Misc' Tab 18
Workspaces and Save 18
The 'DL.BAT' 18
'DL.BAT' for downloading to FlexGate, MINI535 or compatible boards 18
'DL.BAT' for downloading to the MSC121x using the original TI downloader 19
'DL.BAT' for downloading via ATMEL's Flip (batch version) 19

More about C, the ANSI compatibility of µC/51 19
More demos... 20
What's going on? 20

Basic data of µC/51 20
Bug report 20
Data types 20
Integral types 20
Floating point precision 21
Pointer types 21
Memory models 21
8051 memory modifiers 22
Understanding memory modifiers, memory spaces and segmentation 22
Last words about constant strings 25
Memory modifiers in typedefs 25
Register usage 25
Interrupts in C 26
Dealing with call graphs 26
Using the ‘printf()’ formatter 26
A word about strings in general 28
Mixing C and Assembler 28
How to use assembler 29
A very special case for mixing C and Assembler with fixed symbols (using '@') . . . 29
Reentrant functions 30
Indirect functions 30
Variadic functions 30
Integral promotion 30

Old style functions . . . 31
Defining your own SFR's - defining absolute addresses with '@' . . . 31
Overwriting library functions . . . 31
The job of startup() . . . 31
The binary safe '_bin_safe()' . . . 32
Efficient coding . . . 32
Predefined symbols . . . 32
Some important options (command line, #pragma) . . . 32

UmShell & Umake . . . 33
The most important Flags in Make files . . . 34
Make files simple . . . 34
Make files simple, 2.nd . . . 35
Implicit and explicit rules . . . 36
Designing own rules . . . 36

Technical description of the compiler UC51.EXE . . . 36

Technical description of the assembler A51.EXE . . . 37
Mnemonics . . . 37
Names, variables and labels . . . 37
Numbers . . . 37
Operators . . . 38
Directives . . . 38
.segment . . . 38
.include . . . 40
.ibytes . . . 40
.error . . . 40
.end . . . 40
.import . . . 40
.export . . . 40
.file . . . 41
.line - and a word about source level debugging . . . 41
.macro / .endmacro . . . 42
.if / .else / .endif . . . 42
.ifdef / .ifndef . . . 42
.hide / .show . . . 42
.dc.b / .dc.w / .dc.l / .dc.f . . . 43
.ds.b / .ds.w / .ds.l / .ds.f . . . 43
Macros . . . 43
Generated symbols . . . 44

Technical description of the related tools 44

The libraries 45
standard libraries 45
stdio.h 45
string.h 46
ctype.h 47
stdarg,h 47
bin_safe.h 47
math.h 47
8051 specific 47
irq52.h 47
reg51.h, reg52.h, reg535.h, reg552.h, 89C51RD2.h (ATMEL) 48
sys51.h 48
kar.h 48

Appendix A: Migrating from other compilers 48
Memory usage - memory models 48
A useful header file 48
Absolute addresses 49
Interrupts 49
Assembly language 49
Constant strings 49

Appendix B: Distributors 49

Appendix C: Revision history 50
V1.10 50
V1.10.1, V1.10.2 50
V1.10.3 50
V1.10.4 50
V1.10.5 50
V1.10.6 50
V1.10.7 50
V1.10.8 50
V1.10.9 50
V1.10.10 51
V1.10.11 52
V1.10.12 52
V1.10.13 52

V1.10.15 (V1.10.14 omitted) 52
V1.20.20 (V1.10.16-19 omitted) 52
V1.20.01 52
V1.20.02 52
V1.20.03 53
V1.20.04 53

Appendix D: A demo of µC/51's optimiser 53

µC/51 V1.20.04 User's Manual

Preface

Thank you for deciding to use µC/51, a complete ANSI C language development system for the whole 8051 family of microcontrollers.

This is the documentation is for the first "official release" (now in Rev. V1.20). Although this is a version V1.xx, using µC/51 bears no risk of "jumping into cold water": This development system has a long history, the generated code is very stable and highly optimised. Until now, many industrial developments have been done with it. The included source code, libraries and demos cover a broad range: from writing applications in full ANSI C for the smallest available 8051 CPUs (ATMEL's 89C1051 with as little as 1kB of code space and only 64 bytes of internal RAM) up to the largest (µC/51 is able to manage memory sizes up to 16 MB), from Maxim's 1-Wire® Bus, over the I²C Bus, up to Ethernet.

We primarily developed µC/51 for our own, in-house-developments and needs, as a reliable tool. Almost daily µC/51 is used by ourselves for all kind of industrial applications and these test beds are not easy. Besides of that, many useful functions arise, like our I²C Bus library or the 'binary safe' function, that locks a whole binary file against modifications (or can verify the correctness of an update, that was sent over an unreliable line, like radio modems etc.)

History

The idea for writing compilers was born a few years ago. We had written a small BASIC-compiler for use with 8051 based microcontroller. Although this ‘µBASIC/51’ was quite simple, writing software for the 8051’s suddenly became much easier than in Assembler. But with the projects the demands grew too... So the µC/51 ß-Version was developed. Although the ß-Version’s code generator was quite ineffective, it was ideally suited for data logging applications, where speed and software size usually is not crucial, but where other things like floating point math and modular structured software are more important.

Still focusing on ‘small devices’, we decided to embed the Internet in further developments. For this, a high efficient and optimising compiler was needed. This was the trigger for the µC/51 V1.xx. It comes with a totally new code generator, generates a very high efficient code, is easy to use and is full ANSI compatible (with only a very few restrictions due to hardware limitations).

Target processors

µC/51 is suited for all members of the 8051 family. There are no special requirements (like the need of external RAM). We are using µC/51 on ATMEL's 89C1051 as well as on larger systems with banked memory (like our *Flex*Gate (512kB) and or data loggers (currently 2 MB)). Ready-to-start header files and source codes are included for many popular 8051

family members, including some very interesting mixed signal parts, like TI's MSC1210 and the ADuC8xx family from Analog Devices and the devices from SiLabs.

Source code debugging with SiLabs CPUs (formerly Cygnal)

Finding errors in software is difficult. Our older solutions were based on code, downloaded into XRAM, which required special development boards (like our MINI535, MIDI/RS232-80C535, Flash-M1, FlexGate I).

Since V1.20.23 the uC/51 supports a new output format, called „OMF51" . This file format can be used especially with the High-Speed CPUs from SiLabs (Silicon Laboratories) for Source code debugging.

Code quality

µC/51 is based on a universal modelling system, capable of modelling all kind of 8/16/32 bit processor cores. But more important is, that this system was especially designed to deal with 'non linear' architectures (like the 8051's Harvard architecture, with separated code and data space).

µC/51 is the first implementation for this system and the results are very good. There is only a very small part in the software especially dedicated to the 8051, but the generated code can easily compete with the 'market leaders' (as they describe themselves). In some cases, the code is even better (i.e. µC/51 V1.10 translates the SIEVE demo ('SRC\SIEVE\SIEVE.C', which is one of the standard demos) with a module size of 142 bytes, the closest competitor needs 6% more...), the total code size is only 897 bytes - no one of the 'market leaders' does beat this!

Read more about µC/51's optimisation techniques and a comparison with other compilers in the appendix of this documentation.

If demanded (like here) the compiler utilises a 'call graph' scheme for minimum usage of the precious resources (for the 8051 this is undoubted the internal RAM). In combination with a 'data flow optimisation' the amount of allocated internal RAM is astonishing low.

Before publishing µC/51 V1.xx we have implemented (for customers and as a test suite) many reference applications. One was a data logger: controlling a RS485 network of remote sensor nodes. The logger has a built in a cellular (GSM) modem, 2MB Flash memory (for program and data), only (!) 256 Bytes of internal RAM and a Real Time Clock. For minimum power, the modem is hooked to the cellular net only for short times. If required, the logger can send and receive calls and SMS. The complete software required only about 15 kB of code and about 60 Bytes of the internal RAM! No external memory is needed.

TCP/IP Stack and Internet connecticity

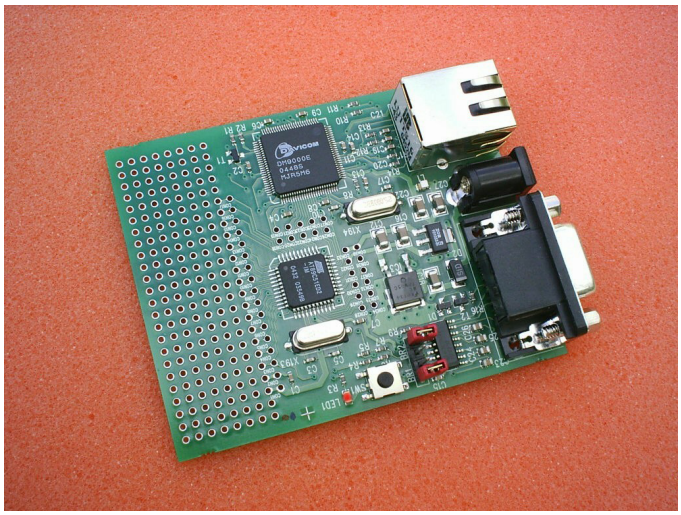
Since V1.20 the uC/51 offers integral support for Internet-based applications;

Accessing the Internet requires a 'TCP/IP stack'. For µC/51 this is no problem, there is no need to spend a lot of money for a 3.rd Party software package, that still requires a lot of time, until it works: uC/51 V1.20 comes with full, integral TCP/IP and Webserver support!

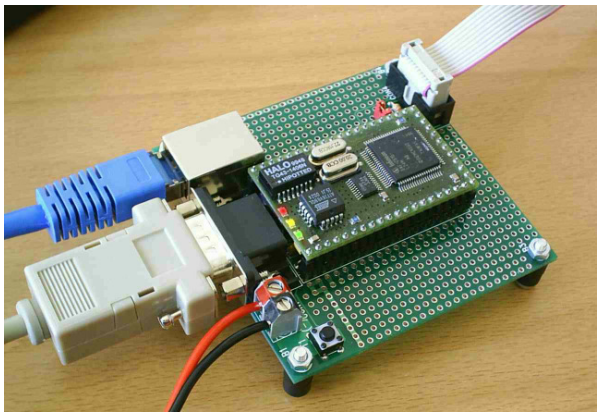
The TCP/IP Stack was primarily designed for our FlexGate(R) modules: match box sized and with a built in 10 or 10/100 MB Ethernet (you may find more information about the 'FlexGate(R) family' under WWW.FLEXGATE.COM and in the directories '...SRC\ED2\' and '...SRC\FLEXGATE\' of the uC/51 installation). The idea for the FlexGates was to have modules with a 'well known infrastructure' on the one side, and a true Web Server on the other side. With only a very low overhead the user can access the application on the FlexGate with a standard Internet browser or by e-mail...

µC/51 comes with an own HTML compiler, hence you can even include HTML, GIF and JPEG files in projecs, as well as accessing C variables in dynamically generated webpages by simply using their name!

Using the TCP/IP Stack is explained in separate documents in the directories '...SRC\ED2\' and '..SRC\FLEXGATE\' (All these documents are included completely since V1.20.01).



Our new **FlexGate III**: 100/10 Mbit Ethernet, Fast CPU, 64kB Flash, 2kB XRAM, 2kB E2Prom, Low-Cost



A **FlexGate I** in action. The core module is about 3x5 cm (out of production, replaced by FlexGate III).

But - returning to the topics - there is still a lot of room for further optimisation in µC/51. We will implement as much as possible.

Limitations of the demo version

The demo version comes with only two restrictions: the maximum size of the generated code is (normally(*)) limited to 8kB. This is more than enough for 'real world applications'. 8 kB allows the use of floating point routines and 'printf()' functions. In one of the demos, shipped with uC/51 we have implemented a complete spectrum analysis in only about 6 kB... The second restriction is: the demo version may only be used for educational or evaluation purposes.

µC/51 does not make any (hidden) changes on the installed computer (like copying DLLs to the system directories, or writing unsolicited entries in the registry...). You can remove it 100% (if using the included uninstaller).

Note: (*) the term 'normally' means, that for some special modules the demo version will allow more code and/or may be used for commercial usage too. For example for the FlexGate I, the demo version can be used for code sizes up to 64kB (this is similar to the full version!), and download to Flash (non volatile code storage) or RAM (fast download and debugging during the development phase). Infos about these exceptions can be found in the appropriate module documentation.

Sourcecode compatibility with other compilers

µC/51 is a full ANSI C compiler. So it will accept any ANSI C compliant sourecode (with only a very few 8051 specific restrictions). However, due to the limitations of the 8051, each manufacturer has made specific extensions to his implementation.

We want to emphasise, that the uC/51 was never designed as clone or 1:1 replacement of existing Compilers. As stated earlier the uC/51 was primarily designed for our own in-house developments.

One of our most important design topics was, to make µC/51 an highly portable and easy-to-use compiler and not yet-another-clone of any other very specific implementation

But in most cases sourcecodes from other (8051) compilers can be compiled without any problems, if only a few items are regarded More information about using sourcecodes from other compilers with uC/51 can be found in the appendix and in the included demos.

Full version

The full version is offered through our web shop and through different distributors. Please visit our website for detailed information.

Support - Versions

A license for µC/51 is valid for all 'minor' releases of this version (here V1). You can download new releases for free from our website. Questions about the actual release should be sent to UC51@WICKENHAEUSER.DE. Support is only possible by e-mail.

You will find a list of improvements/changes of new version (>V1.10) in the supplement.

About us

'Wickenhäuser Elektrotechnik' is a small company, located in the southern part of Germany.

This is our address:

Wickenhäuser Elektrotechnik
Juergen Wickenhaeuser
Nikolaus-Lenau-Str. 20
D-76199 Karlsruhe, Germany
Phone: ++49(0) 721 98849 - 0
Fax: ++49(0) 721 98849 - 29
E-mail: SERVICE@WICKENHAEUSER.DE (for administrative contacts only)
UC51@WICKENHAEUSER.DE (for µC/51 related correspondence)

Your support is wanted

Please support us by placing links to our website:

www.wickenhaeuser.com or www.wickenhaeuser.de

Credits

Many thanks to all the people who have helped us to develop this software. Escpecially to Jens Altmann for the permission to use his JFE - Jen's File Editor.

The future

There are lots of improvements (optimisations, tools, demos, ..) on our list. One is the Graphical User Interface for µC/51. Although the included one is quite sufficient, putting it all under one IDE might be a little more convenient. Additionally we want to expand the *Flex*Gate family by some members, and port the µC to other cores.

Disclaimer

If you disagree in any of the following items, installation and use of this software is not permitted!

Errors

Our software and hardware has been carefully developed and tested. But it is a well known fact, that at the contemporary state of technology, it is not possible to guarantee that a product is completely free of errors. For that reason we decline any liability, loss, or damage caused directly or indirectly by our software and/or use of our hardware. The use is at your own risk.

Trademarks

All terms mentioned in this software and the complementary documentation that are known to be trademarks or service marks have been appropriate marked. But we can not attest the accuracy of this information. Use of a term should not be regarded as affecting the validity of any trademark or service mark.

Internet

All references to Internet addresses are given by best knowledge. However, we want to emphasise particularly that the contents of those addresses usually are beyond our influence. Therefor we dissociate us explicitly from any referenced contents, if they are an offence against any current laws.

Installation and de installation

µC/51 will work on any 'Win9x' (or better) computer. There is no special requirement. This software may be completely removed by the computer's standard de installation routines. Merely files, which have been produced by the use of this software are excluded from de installation. If desired remove them manually.

Important: Please note, that µC/51 must be installed in a path without 'whitespace characters' (i.e. 'C:\\MyFiles\uC51\...', but not 'C:\\My Files\uC51\...'), because µC/51's 'MAKE' system is currently not able to handle such filenames.

Quick start - Desktop symbols

The µC/51 package was designed as a flexible environment. Therefore it consists of several independent programs:

- JFE Jen's File Editor: a powerful and comvenient general Editor
- MakeWiz A Make Wizard for managing Projects (Graphical version)
- µEdit: an easy to use multi file editor with some nice features.

- UmShell: the user interface for µC/51's make system. It will do the job to produce the binary or Hex file out of your sources. Similar function like MakeWiz, but more flexible (and not graphical)
- FlashMon: downloader for systems using the 'OS515' (sourceode included in the demos).
- SLD51: source level debugger (requires 'OS515' and download of code to RAM)

In most cases you will have 'JFE' and 'MakeWiz' or 'µEdit', and'UmShell' and either 'FlashMon', 'SLD51' or a 3.rd party downloader running at the same time.

Obtaining a license for the full version - distributors.

Please regard our distributors! Some distributors offer also the required hardware for development and/or series productions. The current list is found in the appendix.

Installing the full version

If µC/51 is installed, it will run first in the 8kb limited demo mode. If you are owner of a license for the full version, you first have to register your installation. Each license gives your the right to install µC/51 on two (2) different computers, provided that the software is only used on one computer at the same time. From your vendor you will get a 12 digit code, that we will named ‘Key1’ here.

Registering µC/51 is a two step process:

1. First you must start ‘KEY51.EXE' . If a valid license is found for this computer, the data will be displayed, else you must enter the ‘Key1’. With this information ‘KEY51.EXE’ will collect some computer specific data and generate a new key, which has 20 digits (‘Key2’). This ‘Key2’ must be sent to us by e-mail, where it is processed by a machine in short intervals (a few minutes). If you cannot send e-mails from this computer, you must send us the ‘Key2’ manually (as an e-mail).

2. If everything is ok, you will receive a license file with the name ‘UC51.KEY’ as a reply. You must copy this file into the ‘BIN’-file of your µC/51 installation. After that program ‘KEY51.EXE’ should find a valid license, if started again.

Your privacy is not touched! The ‘Key2’ contains only hardware specific information with a strong redundancy, you may change quite a lot of system components before the license file won’t be recognised as valid any more...

If you request more than two license files, you can only order them manually (by e-mail to '`service@wickenhaeuser.de`' and we might ask you for the reason...

We have ensured by independent trustees, that µC/51 can be always distributed, maintained and licensed - no matter what will happen.

Updates - Installing new versions - Keeping a license

We publish new version with small changes on our website at irregular intervals. You may download the latest version at any time. To install the new version, you should uninstall the current version.

Uninstalling will only remove files, that have been installed by the version to be uninstalled. This means, the uninstaller will leave all of your personal files. But if you have made changes in any of the supplied demos, include files or similar then make a copy of your files, or they will be lost without notification...

That is, why we strongly recommend to use own filenames for your own files.

After uninstalling, the new version should be installed to the same path as the previous version, and all will run fine as used before.

Because the license file 'UC51.KEY' is not installed automatically, it will not be touched by the uninstaller, you don't have to take special care about it. Your license will be valid for the new version as well, provided it is the same major version as the previous one.

Transferring a license (New since V1.10.12)

We decided to make installing your personal licenses more easy: If you now once have a valid license file 'UC51.KEY', you an copy it into the BIN directory of your other installation. If you now start the 'KEY51.EXE', it will tell you, that a valid license file has been found - but not for this computer! Then you can simply enter your 12-digit Key1 (which you have obtained from your distributor). If it is correct, it will register the 'UC51.KEY' for this computer.

Please note: We are a very small company and the price of the uC/51 is very low. So please regard the copyrights and the license agreement! Please do not make illegal copies of the uC/51 and the contributed software!

Documentation

The complementary documentation is available as separate files. Please read the documentation carefully prior to the use of the software. You will find the 'latest remarks' in the file 'README.TXT'.

Quick start - a micro tutorial - The NEW V1.20 version!

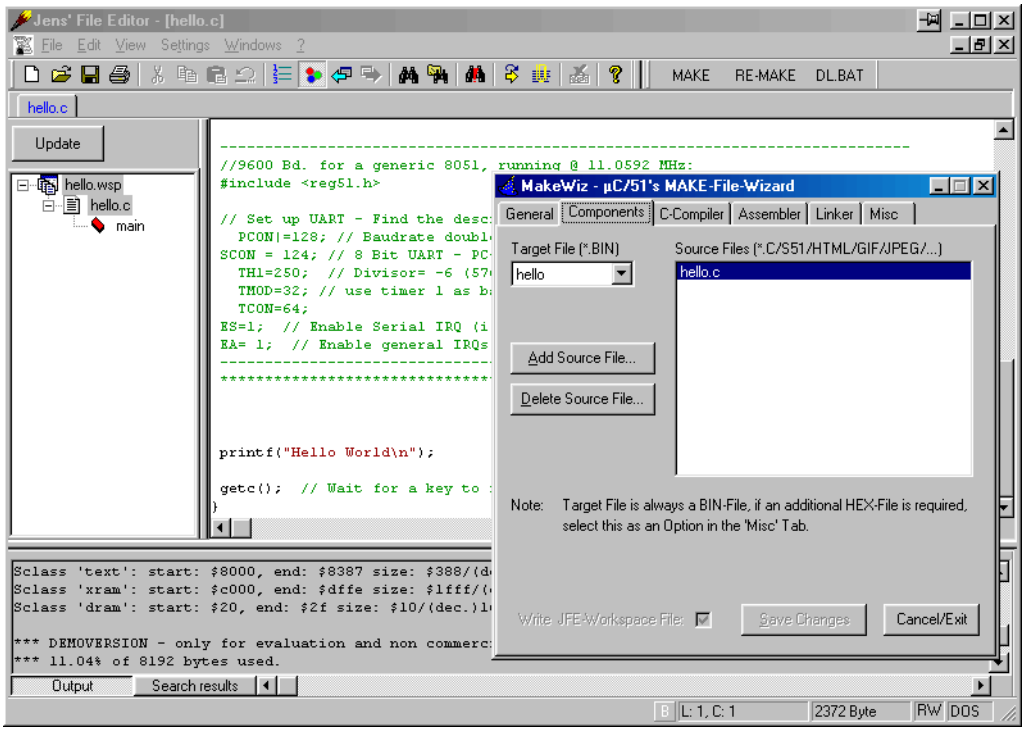
V1.20 comes with a new IDE. But still all other (the 'old') Software is included (feel free to use both version together). The main feature of V1.20 is, that MAKE-Files now are hidden from the user. The complete uC/51 development system is based on these MAKE-Files: Such a file is a recipe, that describes how to build a BINARY or HEX file. MAKE is a very flexible system, but because it is text based, it is a little bit difficult to manage (here with UmShell). Hence we added the MakeWiz since V1.20: This is a GUI for maintaining MAKE

files. MakeWiz has some little restrictions against UmShell, the most important is: It can maintain only Single-Target MAKE-Files.

But most important is: MakeWiz can generate so called "Workspaces" for the JFE (Jen's File Editor). A Workspace consists of several Files and optionally some tool functions. A workspace has the file extension '*.WSP'. But still the Make-Process is based on the MAKE-File. A Workspace may contain less or more Files than the appropriate MAKE-File.

Lets start with the classical "Hello World":

- First start the MakeWiz, then select the file '...\SRC\HELLO\HELLO.MAK'.
- Now make any small change (like inserting a space character in the project remarks) to enable the 'Save Changes' Button.
- Check the Checkbox "Write JFE Workspace" and click on 'Save Changes'.
- Now a Workspace ('...\SRC\HELLO\HELLO.WSP') will be generated.
- Start the JFE Editor and open the new Workspace with "Open Workspace"
- Press 'Make' to generate the Binary File 'HELLO.BIN'.
- Now the screen should look something like that:


Jens' File Editor - [hello.c]
File Edit View Settings Windows ?
MAKE RE-MAKE DL.BAT
hello.c
Update
hello.wsp
hello.c
main
//9600 Bd. for a generic 8051, running @ 11.0592 MHz:
#include <reg51.h>
// Set up UART - Find the desc:
PCON|=128; // Baudrate doubl
SCON = 124; // 8 Bit UART - PC
TH1=250; // Divisor= -6 (57
TMOD=32; // use timer 1 as b
TCON=64;
ES=1; // Enable Serial IRQ (i
EA= 1; // Enable general IRQs
printf("Hello World\n");
getc(); // Wait for a key to
MakeWiz - µC/51's MAKE-File-Wizard
General Components C-Compiler Assembler Linker Misc
Target File (*.BIN)
hello
Source Files (*.C/S51/HTML/GIF/JPEG/...)
hello.c
Add Source File...
Delete Source File...
Note: Target File is always a BIN-File, if an additional HEX-File is required, select this as an Option in the 'Misc' Tab.
Write JFE-Workspace File:
Save Changes
Cancel/Exit
Sclass 'text': start: $8000, end: $8387 size: $388/(d
Sclass 'xram': start: $c000, end: $dffe size: $1fff/(
Sclass 'dram': start: $20, end: $2f size: $10/(dec.)1
*** DEMOVERSION - only for evaluation and non commerc
*** 11.04% of 8192 bytes used.
Output
Search results
L: 1, C: 1
2372 Byte
RW
DOS

The 'HELLO.BIN' is by default compiled and linked for boards, that allow download of software into the RAM (starting at $8000, by using our OS535.BIN on Wickenhaeuser Boards like the FlexGate I). Hence the UART ist not initialised. For a "Hello World" demo for Single Chip CPUs (like the MSC121x Family from Texas Instruments) use '...\SRC\SINGLECH\SC_HELLO.MAK' instead.

That's all. You can now download the generated Binary file, if your board is compatible to OS535.

A MakeWiz generated Workspace contains always three tool-buttons. Note, that JFE will save all changed files prior to executing one of the tools.

'Make': Will generate the target file. This is the normal and fastest way, because only files, that have been changes since the last run will be compiled.

'Re-Make' is required, if files have been changed, that are not explicitly listed in the project (i.e. Header-Files). 'Re-Make' will force a total recompilation of project.

The last Button 'DL.BAT' can be used for starting an own downloader. An example ist the Original Downloader from Texas Instruments for the MSC121x family (named 'download.exe'). The user himself is responsible for the contents of the 'DL.BAT'. The target's file name (without extension) is the one and only parameter for 'DL.BAT'. I.e. for the MSC121x downloader. A 'DL.BAT' for this CPU could look like

```
download /F%1.hex /X11 /P1 /T /B9600
```

(where '%1.hex' will become the name of the HEX file to download, /X11 stands for a aprox. 11 MHz crystal, /P1 stands for 'COM1:' and '/T' with '/B9600' will open a terminal with 9600 Bd. after download..

If you want to change the MAKE-Options, do this in the MakeWiz. If you don't want to write a new Workspace, uncheck the Checkbox.

A closer discussion of MakeWiz's options and the 'DL.BAT' downloader batch file can be found in the next Chapter! Nevertheless we recommend reading the following section about the older V1.10 version. This is, because MakeWiz is still based on the same system, simply hiding the MAKE-files from the user by it's GUI.

Quick start - a micro tutorial - The old V1.10 version!

This chapter will work out **two different ways** to start developing with µC/51:

1. You might have a single-chip controller, a complete single chip controller board or a series production board, which may require HEX- or BIN-files, which include all necessary system initialisation and are linked to start at the address $0000.

2. You have some kind of development hardware available, where software can be donwloaded and executed in an (external) RAM, like our FLEXGATE, MINI535, or compatible, where the program must be linked to a special start address (like for the two above, where the start address for for downlodable programs must be $8000). For downloading a special monitor software is required (we have supplied our monitor OS515). On these boards some initialisation may be omitted (like the UART), because this is already done by the monitor. So you don't have to care about different baudrates or crystal frequencies.
   As an additional advantage you have very small 'turnaround' times (the time between two changes) and a low cost debugging solution (like our SLD51) can be used. If the program

or algorithm is working as expected, it can be used on a single chip controller or series production board as well...

Way 1.): Single chip microcontrollers - series production boards

For single chip controllers we have prepared a few demos in the directory '...\SRC\SINGLECH'.

If you have a suited hardware, you can immediately test them. If not, we recommend to continue reading the next paragraph (Way 2.)).

In this directory are several projects that have been especially designed for single chip controllers. This means, after compiling, the HEX- or BIN-file can directly be programmed into the chip (either by ISP (in-Circuit-Programming), or by simply using a standard programmer.

This projects may be a good starting point for your own developments!

Hardware requirements

You only need a single chip controller with a RS232. The controller should have a crystal frequency, suitable to generate a standard baudrate. For this, we recommend 11.0592 MHz, but others are possible too, like 18.432 MHz, 22.1184 MHz, ...

- For the demo 'KITCLOCK', even a 2kB ATMEL AT89C2051 is sufficient!
- The demo 'A89S8252' requires a AT89S8252, which has an 2kB E2PROM on board.
- All other demos will run fine on a generic 8051 will fit too (AT89C52, AT89S8252, P89C52, even the MSC1210 can be used...).

NOTE: you can not directly connect the RS232 to a controller, because the level is +/- 12 Volt. A level shifter must be used. Either a standard MAX232 compatible part or a discrete solution (as found on our MINI535 module (docu. in '...\SRC\MINI535\M535_V3.PDF')).

First demo: SC_HELLO - a single chip "Hello World"

Usage:

1. Start UmShell and open the makefile 'SC_HELLO.MAK'.

2. If the values for crystal or baudrate differ from your board, press <F2>, correct the values and press <F2> again.

   You can take a look on the C sourcecode: just open 'SC_HELLO.C' with our uEDIT or any other text editor.

3. Now press <F9> to make the HEX- and BIN-file (both contain identical data).

4. Then program your single chip processor with the HEX- or BIN-file.

5. Start a terminal program (we suggest using our FLASHMON[1] : simply check the correct baudrate).

6. Connect the RS232 to the single chip processor and switch power on.
7. Now you should get the "Hello World", each time a key is pressed.

Second demo: KITCLOCK - a kitchen clock

This simple project implements a kitchen clock. Once started, it will count down from 30 downto 0, displaying one value each second (this program is a simplified version of the '...\SRC\MISC\SOFT_RTC.C'). An interrupt is used to generate a precise timing.

1.-6. Same as above

7. Now the terminal program should display the count down. Press a key to see what happens. If you have connected a LED to pin P3.5 (and VCC over a 1-5k resistor), you will see it blinking.

8. Pressing a key will restart the counter.

MENUE/MENUE2 demo - a good starting point for own applications

These two projects implement a small menue system. Because of the usage of floating point and math libraries, the code seems quite large compared to the program size, but feel free to compare ist to Kxxx and Rxxxxxxxxx!

Usage is the same as above.

The MENUE2 adds as an additional feature a timeout to the input function.

A89S8252 demo - accessing the internal E2PROM

This demo requires the very popular AT89S8252. Because of it's internal 2kB E2PROM it can save parameters and user data. This demo shows also the use of assembly language in C functions.

Notes about the MAKEFILES

The linker must start the program from $0000. This is not the default, so it must be set with

```
L51FLAGS = -r0
```

[1] FLASHMON is nothing else than a simple terminal with some additional features

(-r0,0 would be correct too, but since here no external RAM is availabe, it is not necessary)

By default the compiler/assembler will insert breakpoints after each C line. If there is no debugger, this will only give some unwanted extra code. To instruct the Assembler to not include breakpoints, but generating a listing, the macro A51FLAGS must be set:

```
A51FLAGS = -g
```

To make baudrate and crystal frequency a parameter, which can be passed to uC/51, a definition ('-d) is used:

```
C51FLAGS = -dBAUDRATE=9600 -dXTAL=11059200
```

For single chip applications, our debugger SLD51 can not be used. So it is best, to use the fast serial i/o library ('ser_iok.lib' with sourcecode in '...\LIB\LIB_ASS\SER_IOK.S51'). Because this is not the default, it must be set in the makefile:

```
SERIOTYPE=k
```

'ser_iok.lib' requires an UART initialisation with bit TI set!

At last, some rules for the HEX- and BIN-file follow.

Way 2.): Microcontroller hardware - special development boards

As stated before: µC/51 is a development system for all 8051 family members. But for developing and testing, a convenient way is, to use a (special) board, where you can download and run binary files in an external RAM. Of course, there are alternatives, like using an ‘emulator’ or doing a pure software simulation, decide this for your own.

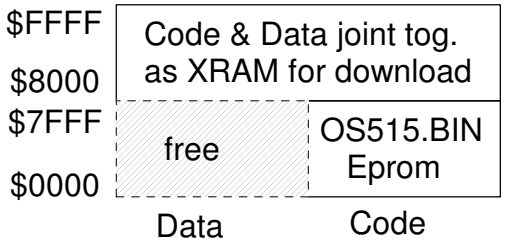
Later, if the program or the tested functions are running as expected, they might be directly used on or programmed to the dedicated target system.

Note: We recommend to use such a special development board for developing functions and algorithms in the RAM. Finally, you may use them on the intended boards, where debugging might be more difficult.

Alternative: Some of our boards (like the FlexGate) offer two different memory models: memory model one allows download to RAM (like the following picture), memory model two offers several different 64kB banks of Flash memory for the code, the RAM is totally free for data. Switching between the two memory models is handled by our 'OS535.BIN', together in combination with a programmable logic chip. That's why they are best suited for development and field use!

Here, we assume that you use a board with the following hardware structure (like our MINI535, FLASH_M1, FlexGate, ...), where you can download the binary files with the

included tools ('FLASHMON' and/or 'SLD51').:



The CPU should be a 80C535, C515, 8051 compatible or a generic 8051. Most of the demos require a 11.0592 to 12.0 MHz crystal. If you have an 80C535 or an C515 with 12.0 MHz, you can start immediately: Burn the file 'SRC\OS\OS515.BIN' to an Eprom (format is 'binary'). For other commonly used CPUs or crystals, the 'OS515.BIN' supports a switch, but nevertheless, it must be rebuilt, as shown below. The board must have a serial port (RS232). If your hardware does have other memory layout, you can easily adapt the 'OS515.BIN' by following the remarks in the source code (and the appropriate make file). 'OS515.BIN' expects a LED on port P3.5 (to let it blink) if it is ready. After a hardware reset, the LED first flashes some times faster, then keeps flashing slower. If an error occurs (data transmission), the LED will flash irregularly. In this case reset the hardware.

Of course, the included demos for the two mixed signal micro controllers, can only be used on their development boards ;-) or similar boards.

Hello World!

Now start 'µEdit' , our enclosed editor. Enter the following:

```
#include <stdio.h>

void main(void){
     printf("Hello World\n");
}
```

Save this text in 'SRC\HELLO\HELLO.C' (might be already there...). Don't close µEdit.

Now start 'UmShell' . This is the second part of the current IDE. UmShell is based on an industrial approved system named 'Make'. For a beginner it might look a little bit strange, but Make is a very powerful tool and in the background of most other IDE's a Make tool is working. Think of Make as just a simple 'recipe' how to build a target. In this case the make file has only one line:

```
Hello.bin: hello.c
```

This tells Make, that 'HELLO.BIN' is to be made from 'HELLO.C'.

By default UmShell knows a set of rules how to translate one extension to another (i.e. *.C to *.BIN). In addition to that Make uses ‘Macros’. Think of them as variables. By default some macros are predefined, i.e. ‘L51FLAGS’ may hold extra settings, to be passed to the linker.

Back to the demo - you may enter the rule above or simply open the prepared make file ‘SRC\HELLO\HELLO.MAK’ in UmShell. Pressing ‘<F9>’ will generate the program. It will have something about 300 bytes in size.

Now start the Soure Level Debugger ‘SLD51’ . Select ‘SRC\HELLO\HELLO.BIN’ for download (maybe you must select the appropriate baud rate for your board first, if it is not 9600 Bd).

After downloading you may start, by clicking on the ‘shoe’ . The program will run in an endless loop printing „Hello World“. You can single step through the assembler commands, C files and set breakpoints with the mouse.

Note: Currently there is no way to step 'over' a breakpoint. First disable it, step over it, then enable it again. After a software stop through a breakpoint, you need two assembler single steps (just press '<F2>' twice). Because are simulated by an 'ljmp' instruction, written to the external RAM. Because 'ljmp' needs 3 Bytes, it is not possible to set a breakpoint everywhere!

For the future we will revise the debugger totally, that it even might be able to debug programs in Flash memory.

Important: Most of the demo makefiles will generate a pure binary file (suitable to be with FLASHMON or SLD51). Other 3.rd party downloaders might only support Hex-files. To instruct the UmShell to make a Hex-file (from the always generated binary file), simply add a new line in the makefile. Here, the additional line would be:

```
hello.hex: hello.bin
```

More infos about makefiles is given in one of the following paragraphs.

MakeWiz - a closer description

MakeWiz was designed, to manage Make-Files. In the old V1.10 version Make-Files had to been loaded, managed and executed with UmShell. Although UmShell is a quite versatile tool, it is more convenient for most users to use a GUI instead of a textual MAKE-File.

By adding the JFE Editor to the V1.20, the UmShell can be omitted for most projects (exceptions are Multi-Target-Projects, Projects with own Rules and Libraries).

This chapter will describe all the MakWiz parameters.

The 'General' Tab

Here you can select existing MAKE-Files or create a new one. The 'Project Remarks' will be added as comments in the MAKE-File.

The 'Components' Tab

A project will consist of at least one Source file. Source files are files, that generate code. Other files might be Header-Files (such as '*.DEF', '*.H' or '*.INC') that generate no code. In the left combo box, one of the Source file names can be selected as the target name.

A binary file for the target('*.BIN' will always be generated). If required, an additional HEX file will be generated (see Tab 'Misc').

The 'C-Compiler' Tab

The CPU Speed is the time (in nsec), a generic 8051 will need for one instruction cycle. For 12 MHz crystal this is 1000. Cores with high speed cores (such as the MSC121x) may vary here, the required value must be estimated.

The CPU Speed is required for the '_wait_ms()' function and (very raw) for the I2C-Bus delay loops.

Additional arguments can be passed to the Compiler in the next line. An example is the definition of an external macro (like '-dAD515' is the same as having a '#define AD515' in the C code, or '-dBAUDRATE=9600' is the same as '#define BAUDRATE 9600'). Seperate the additional arguments by a Space-character.

The lower three combo boxes control the optimisation, memory model and Debug infos. The different memoyr models (small and large) are described in one of the following chapters. We recommend the 'small' model, because it is the fastest and most compact. The 'large' will only be required for very large programs.

The 'Compiler Debug Level' is equal to the Compilers parameter '-g'. This information is primarily required for writing a listing. It will seldom generate larger code (in some rare cases the option 'No debug info' could spare some extra bytes). The last option 'Variables and C Source lines' will be required for the coming Flash-SLD51 in V1.30...

The 'Assembler' Tab

The first Checkbox ('Expand Debug Code') controls, whether the Assembler will include the a macro '__line' after each source line. This works fine with the old Debugger SLD51, but is only unnecessary code ballast, if you don't have RAM to download the software. The new Flash-SLD51 will use this feature too, but currently don't check it.

A listing will include the C-Source if the Checkbox is checked and 'Compiler Debug Level' is not set to 'No debug info'.

The Assembler can have special additional arguments too, pass them in the last Line.

The 'Linker' Tab

The first parameters give the starting address for Code and external RAM. For download to RAM both sections share the same physical RAM, so 8000 with F000 will reserve 4096 Bytes of RAM and 28672 Bytes for the code.

Currently three serial I/O-Libraries exist (Sourcecode in '...\LIB\LIB_ASS').

'Ser_ioD': This Library was designed for boards with code download into RAM by OS535 (like Flexgate, MINI535, MIDI/RS232, ...). It can call the OS535 from within the Interrupt. If this options is used on on of the decribed boards, it can completely be managed remotely over RS232 (like resetting the board, downloading and starting new software). This library uses an interrupt and one byte for a temporary variable), the 'D' stands for Debugger.

'Ser_ioP': This is a compatible, but smaller version, which does not support calls to OS535. Neither an interrupt, nor any additional temporaries are used ('P' stands for Polling).

Both, 'Ser_ioD' and 'ser_ioP' require, that the RI/TI-Bit (in the SCON register) are always clear. About the initialisation of the UART for these modes see '...\SRC\HELLO\HELLO.C':

```
// Set up UART - 9600 Bd for a generic 8051 @ 11.0592 MHz
PCON|=128; // Baudrate double
SCON = 124; // 8 Bit UART - PC-compatible
TH1=250;  // Divisor= -6 (57600/6=9600 Bd.)
TMOD=32; // use timer 1 as baudrate generator
TCON=64;
ES=1;  // Enable Serial IRQ (*** only if required and for Ser_ioD ***)
EA= 1;  // Enable general IRQs (*** only if required and for Ser_ioD ***)
```

The last version 'Ser_ioK' is a another polled Version. But this version requires explicitly that the serial interrupt is not enabled, because the TI-Flag must always be set. The advantage against the previous versions is, that the CPU must not wait, until a character is transmitted.

This is the initialisation from '...\SRC\SINGLECH\SG_HELLO.C':

```
// Set up UART - 9600 Bd for a generic 8051 @ 11.0592 MHz
PCON|=128;       // Baudrate double
SCON = 126;      // 8 Bit UART - PC-compatible, TI-FLAG set!
TH1=250;  // Divisor= -6 (57600/6=9600 Bd.)
TMOD=32;// use timer 1 as baudrate generator
TCON=64;// dto.
```

Of course you can write your own Serial IO-driver, as stated below: Simply choose the same function names and your functions will be used and all libraries will be ignored.

Additional linker arguments can be specified as used.

The last entry allows additional OBJ-Files and additional Libraries: Add additinal OBJ-Files simply by their names, libraries must have a leading '-l'. An example can be found in '...\SRC\LCD2\LCD2_I2C.MAK", where the I2C-Library is added for the demo.

The 'Misc' Tab

If 'Use Smart Printf' is checked, a tailored 'printf()' function will be added. I.e. if there are no floating point formats used, the code for printing floating points will not be added. Uncheck, if not desired.

Always a BIN-File will be written. To convert it into a HEX-File, check the Box. If the HEX File must be downloaded to a Flash memory, the 2.nd Checkbox can be enabled. The all records containing $FF are omitted. This is, because almost all Flash memories must will automatically hold $FF after erasing. Check this option for download to MSC121x, ATMELAT89CC51xx, ADuC8xx and others.

Instead of HEX-Files (which do not contain any debugging information, it is possible to generate OMF51-Format. This format contains the same information than a HEX-File with additionally source code information. The OMF51-Format was especially added for use with SiLabe CPUs.

Workspaces and Save

After all changes have been made, a new MAKE-File must be written. Normally a JFE-Workspace must only be written once, because the tool buttons ('Make', 'Re-Make') automatically refer to the MAKE-File.

The 'DL.BAT'

This batch file is always added to a JFE workspace. It is intended to use the 'DL.BAT' if downloading to target board is possible. The 'DL.BAT' is called with two parameters: The first is the pure target name (without file extension), the second is the path of the project.

Here are three examples for own DL.BAT

'DL.BAT' for downloading to FlexGate, MINI535 or compatible boards

Here the 'Flashmon' will do the whole job. Flashmon can have up to three parameters:

- The FILENAME to download
- -cXX, where XX is the COMXX-port
- -bYYYY, where is the baudrate

If a parameter is missing, its value is taken from FlashMon's INI-File.

```
REM *** Download to FlexGate (into RAM) %1 %2 ***
REM *** Please note:
REM *** Parameter %1: Target Filename without Extension
REM *** Parameter %2: Project Path

REM *** For use with FlashMon

REM *** FlashMon Parameters:
REM -cNR   select COM NR
REM -bBAUD select Baudrate BAUD
REM (here COM taken from INI-File)
REM FlashMon is two directories above and one down from here
..\..\bin\flashmon %1.bin -b115200    (115200 is a FlexGate 1, MINI535 needs 9600)
```

'DL.BAT' for downloading to the MSC121x using the original TI downloader

Here the original downloader from TI must be installed. The programming mode is entered by asserting a RESET signal, while the PSEN pin is pulled low (over a 1k resistor!). The TI downloader can assert this sequence, but if there is no hardware support, the user must enter this state manually befor starting the 'DL.BAT'.

```
REM *** Download %1 (Path: %2)***
REM *** Please note:
REM *** original TI Downloader must be installed!

REM *** TI Downloader Parameters:
REM /F'filname'  /X'nearest_crystal_frq_in_Mhz'
REM /P'COM_nr' /B'terminal_baudrate'

REM *** TI Downloader Switches:
REM /T :open Terminal after download  /H :erase HCR-register

download /F%1.hex /X11 /P1 /T /B9600
```

'DL.BAT' for downloading via ATMEL's Flip (batch version)

The ATMEL AT89C51RD2/ED2 etc. uses a system similar to the MSC121x. Here it is called "autoisp".

```
REM *** DL.BAT for the AT89C51ED2 ****
"C:\Programme\ATMEL\FLIP 2.2.4\bin\batchIsp.exe" -device AT89C51ED2
-autoisp 0 0 -operation onfail abort erase f loadbuffer "%2\%1.hex"
program verify
@if ERRORLEVEL 1 goto :error
REM *** Flashom used as Terminal, COM and Baudrate unchanged
flashmon
@goto :exit
:error
REM *** ERROR! ***
REM *** ERROR! ***
REM *** ERROR! ***
:exit
exit
```

More about C, the ANSI compatibility of µC/51

To learn more about the C language, we recommend to contact a bookstore or a public library. Another endless source of information is the Internet. Many primers and interesting articles may be found. For the future a step by step manual - ideal for educational purposes - will be available.

We have designed the µC/51 for easy use and we have tried to get optimal compatibility with ANSI C. Of course, on a micro controller you won't find functions for disk access or similar. But if you keep in mind, that a 8051 has only very limited resources, you could easily develop algorithms on your PC and then later transfer the software to the 8051.

Large parts of the floating point libraries and other stuff of µC/51 have been designed with Borland C++ 5, the source code may be compiled with Borland C++ as well as with µC/51. Because of µC/51's optimiser, the generated code size is only slightly more than if coded by

hand in Assembler. Another example of portable code is the demo ‘SRC\DFHT\DFHT.C’ for which the PC’s EXE is included... We think it is the best solution to write (library functions) in C and then optimise the compiler to produce ‘handcrafted’ quality. Although it is more difficult at first, it will pay of, because the optimiser will surely reuse, what he learnt by this...

More demos...

In ‘SOURCES’ some more demos are found. All of them can be compiled by loading their Make file in UmShell. The folder ‘SRC\A51’ does contain only one Make file for different projects: select the target in the selection box of UmShell.

What's going on?

Now, as you know how to write a C program, it might be interesting to track it's way to a binary..

In the first step all C files are translated by the compiler 'UC51.EXE'. The compiler may require some system include files (usually 'INCLUDE'*.*'). The compiler produces a plain assembler source file (*.S51). Immediately after that, the Assembler translates the compiler's output to an object file (*.OBJ). So the first step is the transition from *.C to *.OBJ.

For the next step the linker takes all object files at once and puts them together to a binary file (*.BIN). If there are still references left open, the linker tries to get those missing references from the supported libraries (*.LIB) (you may pass as many libraries as you want to the linker). Together with the binary file a listing (*.LST) might be written and a memory map file (*.MEM). Both are intended to be read by debuggers (or the user).

Basic data of µC/51

Bug report

If you think of having found a bug, please let us know! We will try as fast as possible to solve it. Please include a small source to track the bug. Please send bug reports only to `UC51@WICKENHAEUSER.DE`.

Data types

µC/51 is offers the following data types:

Integral types

char	8 Bit 0..255 (chars are treated as 'unsigned', this is ANSI compatible
unsigned char	(as above)
signed char	8 Bit -128..+127 (please note: as a constant -128 is int (ANSI))!
short	16 Bit: -32768..32767 (please note: as a constant -32768 is long (ANSI))!

int	(as above)
unsigned int	16 Bit 0..65535
long	32 Bit -214783648..2147483647
unsigned long	32 Bit 0..4294967295
float	32 Bit IEEE format
char unsigned bit	1 Bit (the modifier 'bit' is no ANSI keyword)

all data is represented in the 'big endian' format. Data types not mentioned here are mapped to the next smaller types (like double to float, long long to long, ...)

The data type ‘bit’ is 8051 specific. It has the values only 0 and non-0 (the integral representation is 1, but the integral value 123 is seen as non-0 too)...

Floating point precision

The floating point routines are based on IEEE proceedings. Rounding is performed as proposed. All mathematical library functions (like 'sin()', ...) have been implemented by using industrial approved algorithms and approximations.

Pointer types

'generic' (far) pointer	32 Bit (explained later, may point to everything)
xdata pointer	16 Bit (external ram)
code pointer	16 Bit (code memory)
near pointer	8 Bit (internal RAM, address 0..127)
inear pointer	8 Bit (internal RAM, address 0..255)
bit pointer	not an allowed type

Memory models

uC/51 supports two memory models: small and large. in the small model all **local** variables are located in the internal RAM ('near'). This is the fastest model. Unfortunately there are only 128 Bytes. If large blocks must be allocated as local variables, the large model must be used. Currently it is not possible to mix both models in one application. Both memory models use the call graph scheme for minimum RAM usage.

The type of the memory which is used for the **local** variables and parameters is the **only** difference for the two memory models! **Global** variables can be placed in both models anywhere! By default (if nothing else is specified) they will be placed in the external RAM or in the code area (if it is a constant). But if using the memory modifies ('xdata' 'code' 'near' 'inear' and bit) each variable can separately been placed anywhere you like! Of course, generic pointers can address the whole 256 * 16 MB address space in both models.

The names 'small' and 'large' denote only the size of the available memory for the local variables. Nothing else.

By default the compilers chooses the small model. To use the large model, the flag 'C51FLAGS' must be set to 'C51FLAGS = large' in UmShell.

Please note: for uC/51 the memory model has no affect on where global variables are located (unlike as in other compilers). We suggest, that you should use a 'typedef' or a '#define' to modify the memory type of global data, depending on the memory model.

8051 memory modifiers

To distinguish between the different types of memory on the 8051 uC/51 defines some new keywords (modifiers):

bit as 'unsigned char bit' this data type uses only 1 bit. Ideal for global flags. The 8051 allows up to 128 bits for data. 128 bits are reserved for 'Special Function Bits' (SFB). Many of them have commonly used names, like 'RI'.

xdata data with this modifier is located in the external ram, access to it needs a 16 Bit pointer

code data is in the code memory, access by a 16 Bit pointer

near data is in the internal RAM, locations 0..127, access by an 8 Bit pointer

inear data is in the internal RAM, locations 0..255, access by an 8 Bit pointer. Using inear is less effective than near, because access is always done by an indirection. It is well suited i.e. for global arrays in the internal RAM. The advantage for inear is, that it might be located in the upper half of the internal RAM, whereas near (and all local variables in the small memory model) must be located in the lower half. If the internal space is low on the controller, consider the use of inear.

far or 'generic' a far modified pointer may point to everything, Currently far pointer are only 'virtual' pointers. You can not declare a variable as far, this will be possible in future version. In some cases the far keyword may be left out, such pointers are called 'generic' pointers (the same is for 'local' data types, because there is no explicit modifier).

Understanding memory modifiers, memory spaces and segmentation

On common PCs there is no need for memory modifiers, because all data can be accessed over a single (usually 32 Bit) pointer type, the memory is 'flat'. A few years ago, in the days of the 'good (?)' old 16 Bit PCs, modifiers were quite common. Even in some header files they still live on (although without any effect...).

For µC/51 we implemented modifiers as an 'option for the wise': If you do not want to deal with memory modifiers: just don't use them. The price you have to pay is code size, memory usage and speed. An example: the Dhrystone demo ('SRC\DHRY\DHRY.C') which is one of

the standard demos of ancient times, does not use them. Nevertheless µC/51 compiles it. But if you let it run, it will be quite slower than it could be. Some competitors offer additional memory models, which are better suited for such kind of source code, but they usually are of limited practical use, because they have other disadvantages (*) and lesser portability...
For an explanation of memory modifiers, let's start without them and look at the following code. If you don't want to take an as close look at the internals, don't worry. The simple usage of uC/51 does not require this:

```
int a;                              // integer in xram
char pre_name[]="Bart";             // constant string, placed in xdata
code char last_name[]="Simpson"; // constant string, placed in code
void say_hi(char *pc, char *pd){
     printf("Hello %s %s\n",pc, pd);
}
void do_it(void){
     char no[1];                    // local array, placed in xram or
                                    // near, depending on memory model
     no[0]=0;
     say_hi(pre_name,last_name);
     say_hi(no,"King Kong");
}
```

'a', 'pre_name' and 'last_name' are so called 'top-level definitions'. They represent the data itself. 'a' and 'pre_name' are global variables with read and write access. uC/51 will place them in the external RAM ('xdata'). 'last_name' is declared as 'code'. The compiler knows, that it might only be read later. So it is safe to locate it in 'code', the Code space (some other compilers offer memory models, that blindly pack everything in the external RAM, only that they can later use single 16 Bit pointers, what we think is an incredible waste of RAM (see (*) above)).

Same is for the String "King Kong". It is definitely a constant string, so it will be placed in the 'code' Code space too. The 'no' is local, there is no choice for placement: in the small model it will become a 'near', in the large model it will become a 'xdata'. It is not 'top-level' too.

So 'pc' and 'pd' might point to anything in the 8051's code space. Because they are pointers and not bound to real data, they are not 'top-level'. This means, the compiler can not automatically determine the type of memory they are pointing to. That is, why they become 'generic' pointers with a size of 32 Bit. By calling the function 'say_hi' uC/51 automatically extends a given pointer to a generic one if needed.

You might recognise, that there is a 8 Bit overhead in the generic pointers? Well, you're right, but we thought, that this extra byte does not hurt more than it might be worth: With such a 32 Bit generic pointer you can access 256 different types of memory, each with a linear size of up to 16 MB! We think this is 'almost' flat too... On some of our boards, we already use this extra byte, i.e. the FlexGate, which has (V1.0) 512kB of Flash memory or the data loggers with 2MB.

The access over generic pointers is done through library functions. You may find their source code in 'LIB\LIB_ASS\MEM32.S51'. Feel free to add your own memory space. After that you must rebuild the libraries, as described later in this document. For our data loggers, we have

mapped almost everything in this space: the Flash with 2MB, internal RAM, even the I²C-peripherals have their own 'selector'...

Summary: All unmodified constant 'top-level' integral data and strings will be placed in 'code', all other unmodified 'top-level' data will be paced in 'xram'. All unmodified pointers will become 'generic' ones.

And what about this (as a global definition):

```
char* pc="HELLO";  // generic pointer, located in xram, initialised
```

This is really difficult! "HELLO" is a constant string. OK. So it will be located in the 'code' space, but what about 'pc' ?. Well, it is definitely not constant, because pc can be written!
So µC/51 will spend a 'generic pointer' (3 byte) in the 'xdata' for it.. The pointer will be initialised by uC/51's startup function, to point to "HELLO". Even if you qualify it as:

```
const char* pc="HELLO"; // generic pointer, located in xram, init.
```

a 'generic' pointer will be used. This is, because the 'const' is not 'top level', which means, it might be changed by runtime to point to a constant object in other memory types (like 'xdata' or 'near'). So the memory type must be included, requiring a 'generic' pointer.

But if you qualify it as

```
code char* pc="HELLO";  // code pointer, located in xram, initialised
```

µC/51 knows, that pc will always point to code, so a 'code pointer' (2 bytes) is sufficient, the same is for:

```
xdata char* pc="HELLO"; // xdata pointer, located in xdata, init.
```

For an explicit memory determination you may use the memory modifiers:

```
xdata int a;  // a is xdata (this is the dafault too)
near char u;  // u is near, access is fastest!
inear char x; // x is in the inear, access is a little bit slower
inear buf[50];// if more internal memory is needed, inear is the
              // best choice
unsigned char flag motor_state;  // On or off
code float[3]={2.23,3.45,5.99}; // a look-up table
```

Memory modifiers for pointers follow the same rules as the ANSI 'const' and 'volatile' modifiers:

```
code char* code pc="HELLO"; // This will fix pc as a bullet proof
                            // 16 Bit code pointer, pointing to code
xdata char* near xp;        // xp is in 'near' but *xp is 'xdata'
far char* fp;               // fp is in 'xdata' (default), the 'far'
                            // is obsolete, but may be more readable
```

Summary: Modifying the memory type in some situation will result in an optimised memory access. µC/51 will track the modifiers over a random number of indirection's. But don't exaggerate it, the compiler will have the better overview than the user...

Last words about constant strings

Mostly the usage of memory modifiers is quite easy. Constructing 'difficult' types, like initialised pointers, pointing to objects in different memory types is not very common in microcontroller programming...

But very often constant strings are used. In µC/51, a constant string should be defined as follows:

```
const char pc[]="HELLO"; // string, located in code, ANSI portable!
code char pc[]="HELLO";  // string, located in code, 8051 recommended!
```

We recommend to use the first version, because it is more portable, The result is the same for both!

Note: Other 8051-compilers (mainly the "market leaders") allow constant strings as

```
code char* pc="Oops!";     // some other compilers! Attention!
```

Although this might be, what the user wanted, this is logically not correct, because 'pc' is treated as an array and not as a pointer. It becomes obvious, if the user really wants to use an initialised pointer. Then the definitions raises an error on these other compilers, like:

```
code char* xdata pc="Oops!";     // Won't work some other compilers!
```

Memory modifiers in typedefs

This is not recommended! There is no situation, where a memory modifier can not be put outside a typedef.

Register usage

On the 8051 we have 4 register banks. By default uC/51 uses the lower two banks (0 and 1). If not desired (as it might be for interrupt functions), the compiler can be instructed only to use bank 0. An example for this can be found in 'SRC\MISC\SOFT_RTC.C'. Bank 2 and 3 are completely free. We suggest to reserve them for debuggers, protocol stacks and other background stuff.

Of course, they can be used as regular space in the 'near' memory. But currently the linker can not do this automatically, because it can not locate 'near' segments below the 'bit' segment. For this, the easiest way is to partition those (at maximum 16 Bytes) manually, i. e.:

```
near char buf_a[8] @ 16;
near char buf_b[8] @ 24;
```

This will reserve two buffers in the bank 2 and 3 area (the '@' directive is described later).

Important: 'startup()' will not clear any register banks. So if using register bank 2 and/or 3 as shown above, they must be initialised, if necessary! On the other hand, this feature could be useful in some cases (i.e. for 'persistent variables', 'magic values' or other things which should not be touched by a (intended or unintended) reset).

Interrupts in C

Interrupts may be written in C. µC/51 is keeping a list of the trashed registers during a function and will only save the trashed registers. To declare a function an interrupts function, the 'interrupt' keyword must be added after the declaration. The function must have a prototype! To assign the function to an interrupts, the macro 'IRQ_VECTOR(function, cpu_irqaddress)' from '<irq52.h>' is needed. See 'SRC\MISC\SOFT_RTC.C' as an example.

Inerrupts at program start 'main()': uC/51's startup-function (see '\LIB\LIB_C\STARTUP.C') disables all interrupts by and-ing $90 to the sfr 'IE'. This disables IRQs in general and all individual IRQ sources of the generic 8051, except the serial IRQ, if it was enabled. This might have come from using a downloader to transfer the software (like our FLASHMON), which requires IRQs for debugging. For the FLASHMON: normally the user will not recognise the enabled IRQ in his software, because the serial IRQs are cought by the FLASHMON (instead of jumping to the mirrored user's software in RAM). For this the F0-flag in the PSW sfr (PSW.5) is used. Some other compiler (like KXXX) clear this flag at program start (although there is no reason to do this). So downloaded programs from other compilers might receive serial IRQs (even if they have no vector installed), which is likely to stall the software...

Dealing with call graphs

Call graphs are graphs, which describe the logical hierarchy of a program. Because each program begins with a 'main()' function, main is always a 'root' of the call graph. After the call graph is built, the linker knows about the used local space. An example: if main() calls two functions 'a()' and 'b()', both may share the same local data (if none of them calls each other). In other situations they may only share parts of the data or none at all. This information is represented by the call graph.

Call graphs can either be totally exclusive or identical. Reasons for having more than one call graph in a program are interrupts or Assembler called C functions. If the linker detects irregularities in the call graph it is not able to build the binary!

The depth of a call graph will represent the amount of used stack! On the a 8051 each call takes 2 bytes, so the depth of the call graph is about the same as the required stack space (this is why each call takes two levels in the call graph).

Of course, the compiler will at first try to use registers for parameters and variables. Only if more space is needed, local variables are allocated!

Using the ‘printf()’ formatter

Formatted output is a too precious routine to waste it to a single 'printf()' implementation! For this our 'printf()' simply calls a formatter function with its arguments and an additional output

function. By using this mechanism it is very easy to generate a formatted output to almost everywhere. One example is 'SRC\RS232_2\RS232_C2.C', where a function 'com2_printf()' is defined with only a few lines. On our data loggers, we use a 'flash_printf()' to write formatted messages to the Flash memory. For our LC Displays a 'lcd_printf()' is supported. For µC/51's libraries the 'sprintf()' functions is based on this formatter!

You can find the complete source code of the formatter ('_doprnt()') in the file 'LIB\LIB_C\DOPRNT.C'. Feel free to modify it! The system is, to pass a character handler, the format string, and the caller's arguments to the formatter.

Currently the 'printf()'-formatter accepts formatting instructions of the type:

% [flags] [width] [.prec] [l | L] type_char

[flags] (optional):	-	Left-justifies the result, pads on the right with blanks. If not given, it right-justifies the result, pads on the left with zeros or blanks.
	+	Signed conversion results always begin with a plus (+) or minus (-) sign.
	blank	If value is nonnegative, the output begins with a blank instead of a plus; negative values still begin with a minus.
[width] (optional):		Minimum number of characters to print, padding with blanks or zeros
[prec] (optional):		Maximum number of characters to print; for integers, minimum number of digits to print.
[l | L] (optional):		Treat the following type as 32 bit (instead of 16 bit)
type_char:	u	Unsigned format
	d	Signed format
	x	Hex format, lower case letters
	c	Single character
	X	Hex format, upper case letters
	f	Floating point format (form [-]dddd.dddd)
	e	Scientific format (form [-]d.dddd e[+/-]ddd)
	E	Scientific format (form [-]d.dddd E[+/-]ddd)
	g	Same as e
	G	Same as g
	s	A string
	%	the '%' itself

As you see, the formatter is quite complete.

The formatter will always return the number of written bytes (= number of calls of for the character handler).

Note: for UART output, each newline ('\n') is replaced by the sequence ('\r\n'). This is required for most terminal programs (but our FLASHMON and SLD51 are inherent to this...). This is done directly by the character handlers of the serial I/O (see 'LIB\LIB_ASS\SERIOD.S51' and 'LIB\LIB_ASS\SERIOP.S51'). So for the formatter each '\n' counts only as one.

Remark: There is no need of explicitly casting 8 bit sized values to 16 bit values. This is automatically done by uC/51, as the ANSI standard requires! The need of an explicit cast (like other 8051 compilers need it), is not according to the standard.

A word about strings in general

Besides the alphanumeric and other printable characters, you can designate hexadecimal and octal escape sequences in µC/51. These escape sequences are interpreted as ASCII characters, allowing you to use characters outside the printable range (ASCII decimal 20-126). The format of a hexadecimal escape sequence is \x<hexnum>, where <hexnum> is up to 2 hexadecimal digits (0-F). For example, the string "R3" can be written as "\x5233" or "\x52\x33". Octals are a backslash followed by up to three octal digits (\ooo). For example, "R3" in octal could be written "\1223" or "\122\063".

Mixing C and Assembler

Mixing C and Assembler is very easy in µC/51. The compiler itself was designed for easy assembler usage. Up to 8 bytes of arguments are passed in registers!

What's going on:

After reset, the controller will jump to 'main' trough a 'ljmp'. The first thing in 'main()' is to call 'startup()'. This routine will initialize the global data of the program and initialize the stack pointer. Then it will return to 'main()'. After 'main()' is done, 'startup()' will be called again...

So if you want to call assembler functions from C, you will find the controller's infra structure well prepared. You even can even use this to get assembler data initialised by 'startup()' (as we did in some demo and library functions).

Because the user might define variable of the same name as CPU registers (like 'A' and 'B'), we decided to user a leading '_' in all compiler exported definitions. A call from C to a function 'test()' will result as an 'lcall _test' (the compiler will write an assembler source code as output). The assembler can access all global C variables and vice versa (but don't forget the consider the '_' ...),

Parameter passing in registers is very easy: All parameters are passed in bank 0!
There are 4 groups of each 2 Bytes, sstarting from R7:R6 (L:H) to R1:2 (L:H)
Allocation is done left to right, where R7 is the first.

If 1. Par. is a Byte it is in R7, if it is 2 Bytes it is in R6:R7 (H:L)
The next Parameter will allocate R5 or R4:R5 (H:L)
If a parameter has 4 Bytes it will allocate either R4..R7 (HH..LL) if it is free or R3..R0

Results are passed in R7, R6:R7 (H:L) or R4..R7 (HH..LL).
The assembler function may change all registers and may use bank 1 for temporaries, if it is sure, that the callers above have enabled bank 1 access (which is the default).

If you pass more arguments to a function, they will be stored in local memory, for assembler usage this is not recommended (it requires access to the call graph), in this case it is better to pass a pointer to the assembler function.

How to use assembler

There are two ways to write software in assembler:

The first: put all assembler functions in one source file (extension *.s51), a shown in some demos. The second: embed a block of assembler instructions in an '#asm' / '#endasm' block.of a C file. This has the advantage, that the C Preprocessor may be used too (like in 'SRC\RS232_2\RS232_C2.C". If there is only one line of assembler, you can use the '#asmline' directive. The instruction follow immediately ('#asmline nop') or in the next line ('#asmline' 'nop'). It is possible to include assembler in a C function, but this is not recommended, because it may disturb the optimiser, which sometimes completely reorders the instructions of a function!

A good starting point is: write a skeleton of your functions in C and use the compiler's output.

Our assembler syntax is slightly different from the 'standard': we think, that our syntax is better readable, for information read the chapter about the assembler's technical reference. Of course, the differences concern only assembler directives and number representation, the mnemonics are full compatible to the standard.

A few short hints for using the Assembler/51: continuous blocks of code should be placed in 'segments'. each segment starts with a '.segment NAME [,OPTIONS]' directive. Segments of the same name will be joint by the linker. The OPTIONS may be 'sclass SCLASS', 'org ORG', 'size SIZE', 'notext'. SCLASS is 'dram', 'iram', xram', without SCLASS the segment is 'code'. ORG sets the segment to an absolute position (the linker may add an additional offset due to it's '-rCODE,XRAM' parameter). With 'SIZE' the size of the segment will be fixed. 'notext' is only of use for data segments. If 'notext' is not set, a mirror segment with initialisation data is added in the code segment (this mechanism is used by the 'startup()' routine to initialise the data segments). The labels of a segments can by exported with '.export' or importet with '.import'. A more detailed description for the assembler is in a later chapter.

A very special case for mixing C and Assembler with fixed symbols (using '@')

While working with uC/51 we discovered a counterintuitive problem while mixing assembler and C in conjunction with fixed variables. The background was: for a monitor we needed some fixed variables (located in the unused register bank 3). The driver contains C and assembler. The problem is, that the compiler will never use the variable by name, if a fixed adress is given, this means '`iri_val=123`' will be replaced by '`mov $1E,123`', but for assembly language the programmer would expect '`mov _iri_val,123`' to be correct (because all the names of (global) C variables are exported to assembly with a leading '_'). But here this won't work, because the compiler will not add this definition for a variable, if it already

knows the address... The easiest way for a workaround is, to add the missing definition for this (rare) case by hand:

```
// --- For fixed Adr. in Bank3 used!
extern near uchar iri_val          @ 0x1E; // Last recived Byte if
extern near uchar iri_flag_byte    @ 0x1F; // Ready-Flag, if set: Char available
(used as Byte: IAP)

// Definitions for assembler-usage of fixed symbols
#define _iri_val $1e
#define _iri_flag_byte $1f
```

Reentrant functions

Reentrant functions are seldom required. To learn more about them, see 'MISC\FACULTY.C'. There is only one special case, where reentrant functions are really required, this is, when a function is called indirect as well as direct, like 'putc()': it may be used direct to send single chars to the UART, but it is also used as a parameter for the 'printf()' formatter, which calls it indirect. Our advice is: don't care about reentrant functions, unless the linker explicitly tells you to use them.

On the 8051 reentrant functions are not allowed to have variable argument lists.

The execution speed of reentrant functions is somewhat slower than non reentrant ones, if there are local arguments to be saved, because all reentrant functions share one memory space for local variables and passed non register arguments.

Indirect functions

Indirect functions may be used in the common manner, there is only one limitation: indirect functions are limited to 6 Bytes for arguments.

All indirect functions will become a root of a call graph, so the linker will not share memory for local variables between them.

Variadic functions

Functions with variable argument lists are common C standard, (like 'printf()'). Variadic functions must have a prototye. There is one special exception for arguments: variadic arguments are always promoted to integrals. You need to consider this to access them later...

Integral promotion

The ANSI standard defines the 'integral promotion'. It says, that all calculations smaller than 'int' are extended to 'int' size, and 'float' to 'double', ... On a 8 Bit microcontroller this will produce a huge overhead of code. You can enable the integral promotion for µC/51 too, by default it is disabled. µC/51 uses a 'smart promotion' scheme: it will only extend the size of a result, if it is required! For functions integral promotion is omitted, if a prototype for the function is given.

Old style functions

They may be used in the 'common old way', but all arguments are automatically treated by integral promotion, even if there is a prototype following. Mixing old style and new style declarations in different files may lead to undesired results...

Defining your own SFR's - defining absolute addresses with '@'

One of the reasons for the success of the 8051 family is, that new family members simply have more or other 'special function registers' (SFR). To use a special family member, we have supplied some register definition files like 'INCLUDE\REG51.H' or '..REG52.H' or 'REG535.H'. To extend or modify the given definition files, there are two ways:

The first way might be used, if it is important to have the definition for assembler as well as for C: First add the definition for Assembler (like 'WPM_BIT = P3.4'). Then in a second line this definition must be mapped to the appropriate C symbol, because C adds an '_': '_WPM_BIT = WPM_BIT'). In the last step the new symbol must have a declaration for the compiler 'extern unsigned char bit WPM_BIT;'.

The Second way is more easy, but it declares the new symbol only for C use: simply add the definition to the declaration: 'unsigned char WPM_BIT @ 0x85;'. That's all!

Overwriting library functions

You may overwrite C library functions (those with a '_' as first character...). For instance if you think you have a better 'sin()' function than ours: simply add your own as an *.OBJ file to your program. Symbols in objects files always have a higher priority than library symbols.

If you overwrite an assembler symbol, the linker will use your definition, but it will present you a warning, which you might ignore...

The job of startup()

The 'startup()' function is responsible for the initialisation of the microcontroller's stack and all types of RAM (and even Bits): All variable spaces are either set to 0, or initialised by their initial values. You may customise the startup()-function to perform some extra work (find it in ‘LIB\LIB_C\’ (it is not a real C function, but it requires the C preprocessor).

Sometimes the user might be interested to run one specific routine before doing all the initialising. One example is the C515’s impacient watchdog: It has a timeout of only 65 msec. But if there is a lot of data to initialise, it might take longer than this period. The result would be a constantly triggered watchdog reset. To omit such things, the user can specify a custom function. It is called immediately after reset, but internal RAM is already cleared and the stackpointer is set, so it might be even written in C:

```
// This tells the linker, that a custom function must be called first:
#asm
.export STARTUP_FIRST
STARTUP_FIRST=1
#endasm
```

```
// The function itself: The name is given!
void _startup_first(void){
     ...
}
```

For the C515 our solution was, to start a 60 msec. interrupt, delaying the watchdog to 16 sec... Another idea is to jump directly to '_main', if no initialisation of the external memory must be done.

The binary safe '_bin_safe()'

With this function a program can check it's integrity. We have provided this function, to lock binary files against modifications. An example: one of our customers distributes his binary updates (for his machines) by e-mail, the users can upload the update by themselves. So there might be a chance, that the one or other user tries to make changes in the binaries, like text messages or something else. But the software might still work!
'_bin_safe()' will detect such changes with a very high reliability. Because it does not compute a simple checksum over itself, it uses a 16 bit CRC instead, which is quite hard to bypass ;-).

'_bin_safe()' will return 0, if no changes are found. Decide for yourself, what to do in the other case... Include the header file 'bin_safe.h' to declare the function. Due to the algorithm, the speed is not very fast, on a 12MHz 8051, about 8kB per sec. can be examined.

Remark: If one of our competitors has such a function too, please let us know...

Efficient coding

Keep in mind, that the 8051 is a 8 Bit CPU, whereas most books about C assume a 32 Bit bolt. Unsigned operations usually are significantly faster than signed on the 8051. And the native' data type for an 8051 is not 'int' (as for 16 Bit and 32 Bit CPUs), but 'unsigned char'.

Predefined symbols

The compiler defines some preprocessor symbols, which are common standard (except for "__UC__" and "_i8051"):

__DATE__	today's date (the '__' are two '_')
__TIME__	and time
__FILE__	the name of the source file
__LINE__	and the current line no.
__STDC__	defined as "__STDC__"
__UC__	defined as "__UC__", this is µC/51 specific!
_i8051	defined as 1, this is µC/51 specific! (only one leading '_').

Some important options (command line, #pragma)

The compiler can be controlled either by command line, or by '#pragma' directives. For the command line options, the compiler will show a full list, if called as 'UC51 -?'. Here the most

important options are described. You can pass them to the compiler in a makefile by using the variable 'C51FLAGS'):

-g	Debug level (1 (= default) or 2). If set to 2, the compiler will include more source info in the output, but the code might be less optimised.
-size	Optimise for size, this is the default
-speed	Optimise for speed
-a	Suppress strict warnings
-w	Suppress all warnings
-dMACRO	Define a Macro (to assign a value use '-dMACRO=VALUE')
-b0	Use only register bank 0 (by default 0 and 1 will be used)
-nograph	Disable the call graph. Will need a lot of RAM...
-noline	Don't emit code for the macro '__line'. You can supply an own version for this macro, see 'Technical description of the assembler...'

Some parameters are also available as '#pragma's:

'#pragma option -gX'	Set debug level, X=1 or 2
'#pragma option -a'	Suppress strict warnings
'#pragma option -a-'	Enable strict warnings
'#pragma option -w'	Suppress all warnings
'#pragma option -w-'	Enable all warnings
'#pragma cpu51 -b0'	Use only register bank 0
'#pragma cpu51 -b0-'	Use register bank 0 and 1
'#pragma cpu51 -noline'	Don't emit '.line' directives
'#pragma cpu51 -noline-'	Emit '.line' directives
'#pragma cpu51 -nograph'	Don't emit call graph information
'#pragma cpu51 -nograph-'	Emit call graph information
'#pragma cpu51 -large'	Switch to large memory model
'#pragma cpu51 -small'	Switch to small memory model
'#pragma cpu51 -labeldist X'	Set new label distance for short jumps
'#pragma cpu51 -labeldist-'	Restore default label distance

The label distance is a number (currently set to 50 as default). Because µC/51 has no internal assembler, it does not know, if a short jump could be used as an optimisation. Therefore it uses a guess: Most instructions are less than 2 bytes long, so a label distance of 50 would allow short jumps within a distance of 50 assembler lines (or +/- 100 bytes) which should be safe. However, increasing this value could improve code size. If the assembler requires to decrease it, please let us know.

UmShell & Umake

As introduced beforee: these two helpers will build your software, by using a 'recipe' (also named 'makefile'). UmShell is the graphical user interface, but Umake will do all the work...

The most important Flags in Make files

The flags in the Make file are nothing more than variables, able to hold a string value. As there are:

C51FLAGS: Flags for the compiler, default is empty. My be used for telling the compiler something, like a definition: (‘C51FLAGS = -dTEST -dABC=3’ is the same as if ‘#define TEST’ and ‘#define ABC 3’ inside the source code.

A51FLAGS: Flags for the assembler, predefined as '-d -g'
-d: expand the macro '__line' for each sourceline (needed for single step)
-g: include sourcecode info in the listing

L51FLAGS: Flags for the linker, predefined to '-r$8000,$F000', this will link all programs to $8000, with xdata starting at $F000.

MODEL: The memory model, default is 'small', set to 'large' if required

SIOTYPE: The used library for the serial i/o. Default is ‘d’, for using the interrupt driven one. In some cases, a polled version might fit better (the polled version gets all possible values (0..255) and is smaller in size, but the interrupt driven version allows source code debugging. Set to ‘p’ to use the polled version

PFLAGS: If set to ‘PFLAGS=FULL_PRINTF’ the full featured printf()-formatter ist used. This might be interesting, if format strings are dynamically generated during execution. By default the value is ‘SMART_PRINTF’, which allow the compiler to tailor the formatter to the minimum size.

The default definitions for the macros above are found in 'BIN\BUILTINS.MAK'. If you want to get help about additional parameters, you can start the program (Assembler, Compiler, Linker, ...) from command line with the parameter '-?'.

One of the most important items is 'A51FLAGS -d ...': Usually the compiler writes a definition for the macro '__line' at the beginning of each translated file. Currently this macro does nothing else than make a call to location $0006, where the debugger is expected. The assembler then will expand this macro for each C source line in its output (so you can single step through your program). But if the software development is finished you should disable this feature, because for each line 3 bytes are spend... (if you forget it, this is not crucial, because there is a default handler, so your software won't crash...).

The format of the numbers in the L51FLAGS can be either '0x', decimal, or '$' for base 16.

Make files simple

This is a minimum description for make files. The idea behind make is, to have an recipe of how something is made from something else, if the result is older than the sources. in other words: Make will only do what is really necessary. Make is based on rules, depending on file

name extensions. You may find the default rules in 'BIN\BUILTINS.MAK'. We have supplied rules for the most common tasks here in, like:

Making a *.bin out of *.c or *.s51
Making a *.obj out of *.c or *.s51
Making a *.bin out of *.obj and *.lib
Making a *.hex out of *.bin
Making a *.omf out of *.bin

Using a rule in a make file always starts with the result, followed by a colon. Then the sources follow. To select a rule, Make must detect the Target's filename in the sources. i.e.

```
happy.bin: fast.obj happy.obj newlink.obj
```

If any of the three sources are younger than 'happy.bin', Make will try to find a rule how to make happy.bin. It will detect, that 'happy' is in happy.obj, so it will use the rule for Making *.bin out of *.obj. In case of success, the file time stamps will reflect this.

Guess, what the following line will do:

```
happy.hex: happy.bin
```

or

```
happy.omf: happy.bin
```

In some cases Make is not able to detect all dependencies, i. e. if you change header files for a source code, but Make does not know about this dependency (it could be advised to watch for this, but often one is too lazy to tell that...). In this case you can tell Make to rebuilt the whole project without regarding the dependencies.

The place where a macro is defined, is not important, because first the make file is totally read. It is allowed to overwrite macros or to use other macros as parameters (in brackets with a leading '$'). A simple '$*' is replaced by the filename of the destination (without file extension), a '$<' is replaced by all dependencies (for UmShell this is '<F8>').

Make files simple, 2.nd

Ok, until now makefiles are easy to use, but probably many users might even prefer to have something like 'drag-and-drop' and some forms, where all the settings can be done by clicking.

In a future version we will support UmShell with some kind of 'expert'. This expert will hide the makefile from the user and he will only see some 'project space'. But for the current version, we decided to spend as much time as possible in a high quality compiler and thus keeping the price for the package low.

Implicit and explicit rules

Umake does support two kind of rules: implicit and explicit. Implicit rules are those, that are the same as the default rules. Default rules are declared by the extension:

```
.s51.bin:
            $(A51) -e $*.s51 -i$(INCLUDE) $(A51FLAGS)
            $(L51) -e -o$*.bin $*.obj -l$(STDLIB) $(L51FLAGS) -m -s
```

This is one of the default rules. It will convert any '*.S51' file into a '*.BIN' file. An implicit rule is used by supplying a 'target' and some 'sources'. If one of the sources filename (without the extension) matches, the appropriate implicit rule will be used. I.e. if you want to make a file 'A.BIN' out of 'A.S51' you can simply enter in your makefile:

```
a.bin: a.s51
```

An explicit rule is something, where you supply commands for this special use. I. e. if you have the example from before, but you would like to use another (linking) assembler, simply write:

```
a.bin: a.s51
         my_assli -o$*.bin $*.obj
```

Designing own rules

Because Make is one of the most versatile ways to 'make' something, you can easily expand it by own rules. Think of the following scenario: you have just written a new converter, which is able to generate a OBJ-file out of a HTML-file (might be necessary for an embedded webserver). This program might have the name 'HTML2OBJ' and is called with source and destination file name. The generalize this, an implicit rule could be used:

```
.htm.obj:
         $(HTML2OBJ) -e $*.htm $*.s51       # first produce an assembly file
         $(A51) -e $*.s51 -i$(INCLUDE) $(A51FLAGS) # then make an OBJ out of it
```

Here the macro '$(HTML2OBJ)' is holding the path of ther 'HTML2OBJ.EXE' (see 'BIN\BUILTINS.MAK').

And from now on, you could even link HTML to your application! (That's exactly the way it works since V1.20)

Technical description of the compiler UC51.EXE

This chapter will work out some of the deeper details of the compiler. It will follow in an later version of this documentation...

Technical description of the assembler A51.EXE

The A51 is a macro assembler. It was designed as a reliable and stable workhorse and (as for the compiler) the A51 is almost totally independent from the 8051 too! A51 operates as a single pass assembler. Thus making it very fast.

Mnemonics

The syntax uses the common standard Mnemonics and is sensitive. Mnemonics itself are not case sensitive, but any used symbols (you may write 'mov a,#0' or 'mov a,#0', or 'mov ACC,#0', but not 'mov acc,#0', because ACC is a symbol, defined in an include file). As a convention, all of the 8051's SFRs are upper case letters.

In cases, where a number is expected, you may even use a calculation, like simply writing

'mov R6,a', you could write (could be useful in macros, use braces):

```
WORKREG=5
mov R(WORKREG+1),a
```

Names, variables and labels

All names must be composed as in C: first char: '_', 'a-z' or 'A-Z', rest may contain numbers. A name must not exceed 64 characters, else this will lead to an error message.

Labels have a following colon and are treated as addresses.

A variable is a symbol, that can hold a value (assignment). Multiple assignments for the same symbol are allowed:

```
nr=11
mov A,#nr      ; nr is 11
nr=21
mov A,#21      ; nr is 21
```

Temporary labels may have a '?' as first sign. Temporary labels behave like normal labels, except they will never appear in a listing.

Numbers

The assembler will accept many different formats:

123	regular decimal number
'X'	8 bit number (ASCII character)
'AB'	16 bit number (like ASCII character: ('A'*256)+'B'))
0x100	Hex number (256 dec.) C language syntax
$100	Hex number (256 dec.) Motorola syntax

0100h	Hex number (256 dec.) Intel syntax (version 1)
0100H	Hex number (256 dec.) Intel syntax (version 1)
100h	Hex number (256 dec.) Intel syntax (version 2)
100H	Hex number (256 dec.) Intel syntax (version 2)
0100	Octal (base 8) number (83 dec.) C language syntax
%111	Binary number (7 dec.)
%000111	Binary number (7 dec.)
0111b	Binary number (7 dec.) Intel syntax (version 1)
0111B	Binary number (7 dec.) Intel syntax (version 1)
111b	Binary number (7 dec.) Intel syntax (version 2)
111b	Binary number (7 dec.) Intel syntax (version 2)

We recommend the formats with "$" and "%". They are the most readable ones.

Operators

The A51 uses the same operators and hierarchy as C (except bit addressing)

( )	Braces, highest Priority
.	Bit addressing (like ACC.7: highest bit of ACC)
* / %	Multiplication, division, module
+ -	As sign
<< >>	Shift operators
< > >= <=	Comparison, value is 1 if true, 0 for false
&	Binary and
^	Binary exclusive or
|	Binary or
&&	Logical and
||	Logical or

Directives

.segment

Usage: .segment NAME [, Parameter]$_{opt}$.

Open an existing or a new segment 'NAME'. The linker will collect all segments with the same NAME and treat them as an unit. The compiler will generate a segment for each C function. We recommend to use as NAME the name of the function (or entry label) plus a leading '_'. For functions each function should have its own segment for the code, whereas for data of the same type only one segment should be used. The names for the data segments are given and so the 'startup()' function can initialise them properly (the IRQ driven serial I/O driver ('LIB_ASS\SERIOD.S51') uses this technique to declare a byte variable in the 'nearbss' segment. This segment is cleared by 'startup()'.

On demand, the linker will generate additional symbols, allowing access to internal data of the segments (like size, load address, ...) This feature is normally not used

Note: The linker will remove all unused segments, only referenced segments of segments containing an 'org' parameter will be linked to the binary.

Parameter: org NUMBER

Sets an absolute address for this segment. Only once per segment allowed. The program will start with the first label of the segment with org $0000. This value might be added by and offset from the linker, like for our development boards. The linker offset is set to $8000 by default, where the RAM starts and the program will be downloaded to.

Parameter: sclass NAME

Sets the memory class of the segment. If no 'sclass' is given, the default sclass is 'text'

Allowed NAMEs for the 8051:

text	Code, constants.("EPROM"). This is also the default storage class
xram	The external RAM
bit	The bit field in the internal RAM (from $20..$2F)
dram	The internal, direct accessible RAM (from 0..$7F)
iram	The internal, indirect accessible RAM (from 0..$FF)

Parameter: size NUMBER

Defines the maximum size for this segment. For multiple set 'size': only the largest directive is used. The compiler makes intensive use of this directive to manage local variables.

Parameter: fill

If set, the linker will always set the size of the segment to its maximum size (makes only sense in combination with parameter 'size')

Parameter: notext

Normally there is no way for the linker to initialise non-code segments (like RAM). So the linker puts all initialisations data in a 'mirror segment'. This 'mirror segment' might be accessed by the 'startup()' function to copy it to the RAM,

Parameter: page NUMBER

If this parameter is given, the segment will be completely located inside a page of the given NUMBER. As an example: if the segment uses 'acall' and 'ajmp' instructions, NUMBER must be 2048. A fixed value can be set for the segment (parameter 'size'). Normally for NUMBER only powers of 2 are used.

Parameter: align NUMBER

This parameter specifies the condition for the first byte of the segment. I.e. if a segment must start at an even address, NUMBER must be 2. 'align' and 'page' might be used in any combination.

.include

There are two versions: '.include <FILE>' and '.include "FILE". Both version include a (text) file in the assembly. The difference is: '<FILE>' uses the path, given to the assembler by command line (option '-i'), which points normally to µC/51's 'include' directory, whereas ' "FILE" ' uses the directory relative to the current directory.

Included files may include further files.

.ibytes

This directive will include a pure binary file (i.e. a data table). only the format '.ibytes "FNAME" ' is allowed.

.error

An optional text might follow this directive. An error will be displayed and assembling is stopped. Like:

```
.if VERSION!=1
     .error Only version 1 alloweed"
.endif
```

.end

Assembly is stopped. All following text is ignored (even if '.end' is in an included file!).

.import

Usage: .import SYMBOL [, FURTHER_SYMBOLS]$_{opt.}$

To use a symbol, defined in another sourcecode, it must be imported (otherwise the assembler would give an error). This is done by '.import'.

.export

Usage: .export SYMBOL [, FURTHER_SYMBOLS]$_{opt.}$

Exporting a symbol allows its global use. If a symbol has '.import' and '.export' in the same sourcecode, an '.export' is assumed. This is useful for libraries. There is also the possibility to export assignments in µC/51 (i.e. used for the '_startup_first()' function). This might be useful

i.e. for libraries. I.e.: If the I²C-library would import the SDA and SCL pins (and not define them by itself), the user could write the following:

```
.export SDA,SCL
SDA=P1.7
SCL=P1.6
```

This would set SDA and SCL only for this project to P1.6/7. Other pins could be used for other projects without changing the library!

Note: If dealing with compiler generated symbols, you must consider, that the compiler will add a leading '_' to each symbol. So the function 'test()' in C correspondents to the symbol '_test' in assembler!

.file

The directive '.file "HLL_FILENAME" ' tells the assembler, that a high level language sourcefile (currently only µC/51) corresponds to this assembler file. Only one '.file' directive is allowed in an assembler sourcefile.

.line - and a word about source level debugging

The directive '.line NUMBER' tells the assembler, that all following mnemonics correspondent to line no. NUMBER of the sourcefile, previously set by a '.file' directive'. If the assembler is instructed to produce a listing (commandline '-s') or to include high level sourcecode information in the object file (commandline '-g'), the assembler will include the appropriate high level lines. Otherwise nothing will happen.

But '.line' has a second, very important function: It may be used to lard the binary file with breakpoints: if the assembler is instructed to expand the macro '__line' (by command line's '-d') and a macro with the name '__line' is called, with NUMBER as an argument. By default the µC/51 includes a definition for this macro as 'ljmp $0006' (the argument NUMBER is currently not used). At address $0006 there will be either the 'OS515.BIN' or a simple 'ret' instruction (added by 'LIB\LIB_C\STARTUP.C' as a safety precaution). So, after each high level line (C statement) the monitor OS515.BIN is called.

This is a very nice feature for debugging software. Unfortunately each breakpoint is three bytes of extra code (and a little bit of time). To disable this feature, set either the definition 'A51FLAGS' in the project's makefile (UmShell) to '-g' (listing only) or to something else. The default is 'A51FLAGS = -d -g' (from 'BIN\BUILTINS.MAK').

Remark: We are currently planning a new version of the SLD51. For a 'real breakpoint' SLD51 will modify the 'ljmp $0006' by something else (possible because the program is located in the RAM). This is not possible for Flash memory. Because nowadays many 8051's are equipped with Flash memory, we will modify the breakpoint system to a 'non destructive' system. The new SLD51 will be able to debug on all 8051's, directly on the field used board.

Note: The above mixing of assembly and sourcecode is quite simple. Because the compiler sometimes totally reorders the logical flow of a function, the output might not be 100% synchronise in some cases. This is not an error, it's just a matter of lazy cosmetics...

.macro / .endmacro

Usage: .macro NAME

The A51 is a macro assembler. A macro is a simple text replacement with some additional features. Programing with macros is described in the next part of this documentation.

.if / .else / .endif

Usage: .if CONDITION

This directives are use for conditional assembly. The '.else' part is an option. The condition must be full evaluable at assembly time. You can not use labels for the condition. Conditions might be nested, like:

```
.if DEBUG_LEVEL ==1
     .if SUBVERSION >=2
          lcall test12p
     .else
          lcall test11
     .endif
.endif
```

.ifdef / .ifndef

Usage: .ifdef SYMBOL and .ifndef SYMBOL

This directive simply checks, if a SYMBOL is already defined. It is quite similar to the '.if' above. An example:

```
.ifndef char_out        ; If macro not already defined
     .macro char_out    ; define it!
          mov A,@1      ; parameter @1 into A
          lcall output  ; and OUT
     .endmacro          ; now the macro is definitely defined!
.endif
...
char_out 'X'            ; use the macro for output of a 'X'
```

.hide / .show

This two directives increment and decrement the 'documentation level' of the assembler. Only if this level is greater than zero, a listing or sourcecode information is generated. During expansion of a macro, the level will be temporarily decremented. So if you're interested in the expanded macros, simply add an addition '.show'. The default level is 1.

.dc.b / .dc.w / .dc.l / .dc.f

These directives insert numbers and strings in the output. A comma is used as a delimiter. For '.dc.b' strings may be used. The byte order for '.dc.w' and '.dc.l' is Big Endian, this means highbyte first. The '.dc.f' inserts a number in the IEEE32 32 bit floating point format. Some example:

```
.dc.b  "Hello World",13,10,0       ; Text, <CR>, <LF>, 0
dc.w  init, $0000, 'AB' ; ; 'AB' is $41,$42 (same as .dc.b 'A','B')

.dc.w  test    ; sppose test=$8023, would be same as .dc.b $80, $23
.dc.l  init,  'ABCD'        ; 4 byte constant
.dc.f  3.14159, 2.718282    ; two floating points...
```

.ds.b / .ds.w / .ds.l / .ds.f

Usage: .ds.X NUMBER

The '.ds.X' directives do nothing than simple reserve the number of bytes, words, longs or floats. NUMBER must be a constant value or expression. For the reserved space, bytes of the vale 255 are written. NUMBER might be zero. In this case nothing is written.

Macros

A macro is a simple text replacement with some additional features. A macro may contain other macros and usage of temporary labels.

Before a macro can be used, it has to be defined inside a '.macro' / '.endmacro' block.

Each macro can have up to 10 arguments '@0' to '@9'. The number of supplied arguments is given in '@0', the first argument is in '@1'. If using an undefined argument, an error will be generated. Even complete strings might be used as macro arguments! Please note: during the expansion of a macro, the 'documentation level' is decremented (see '.hide / .show').

Temporary labels are generated for each expansion. An example:

```
.macro help_text   ; help text in a table
   .dc.b @1,@2,0        ; The system : Help-ID, String, 0
.endmacro
...
.segment help_tab, sclass text
help_text 22,"Valve open"             ; usage
help_text 25,"Temperature low"        ; another usage
help_text 17, "(C) Copyright 2003"

.macro safe                  ; save register
   mov R0,@1                      ; start
   mov R1,@2                      ; len
?s0: mov A,@R0          ; not an argument, instead a mnemonic
   inc R0               ; next addr.
   push ACC
   djnz R1,?s0     ; '?s0' treated as '?ID_s0' (ID: 1,2,...)
.endmacro
```

```
...
sichern _temp,17 ; 17 bytes safe from
...
sichern _temp,10 ; 10 Bytes safe from temp
```

The label '?s0' in the both expansions is different.

Generated symbols

On demand (if imported with '.import'), the assembler and the linker will generate additional symbols, containing infos about segments. The system: let "%s" be the segment's name, so these additional labels are available (a double '_' at the beginning and end):

__%_org__	segment's start (real) address
__%s_size__	segment's (real) size in bytes (bits for 'sclass bit')
__%s_maxsize__	maximum segment's size (if supplied)
__%s_load__	address of the initialisation data in the code space (if supplied)

Additionally for each 'sclass' (for "%s"sclass can be "text", "bit", "dram", "iram", "xram"):

__%s_org__
__%s_size__

Last not least:

__stack_org__	the first unused byte in the internal RAM is used for the stack. It will grow upwards!
__bin_size__	this is the complete size of the binary, including all code and initialisation data.

Technical description of the related tools

This chapter will work out some of the deeper details of the related tools. It will follow in an later version of this documentation...

Bin2Hex: Binary (BIN) to HEX converter - This tool allows automatic conversion of BIN files into HEX files. There is an interesting option '-s', which will reduce the the HEX file's size by omitting $FF bytes. This feature is intended to be used for Flash-downloads, like the MSC121x-family (see remark at the comments for the V1.10.10 release). A macro has been added for makefiles: B2HFLAGS. It may be setz to '-s'.

Umake: Make tool driven by UmShell

Hdump: Hex-Dump Tool - command line driven

LibMan: Library manager - command line driven

The libraries

Currently only the most important library functions have been added. But the list is continuously growing...

standard libraries

stdio.h

This is one of the most important header files. All functions work as the standard proposes.

The following two defines simplify the use of the 8051's 'native' data type and are quite common:

```
typedef unsigned int uint;
typedef unsigned char uchar;

void putc(uchar) reentrant; // reentrant because called indirect
uchar getc(void);
uchar kbhit(void);

void putchar(uchar) reentrant; // same as putc and getc...
uchar getchar(void);

// printf() and sprintf() return the no. of characters printed (ANSI)
// read more about the format string in an earlier chapter...
int printf(far char* pfmt, ... ); // print using putc()
int sprintf(far char *dest far char* pfmt, ... ); // print to a string
int puts(far char* ps);          // result is always 1 (ANSI)
```

The following two functions implement a pseudo random number generator, based on a linear congruent algorithm. The sequence will always be the same, depending on 'seed'. A 4 Byte global variable will be used to keep the last result (sourcecode in 'LIB\LIB_C\RAND.C').

```
unsigned int rand(void);     // pseudo random number between 0..65535
// Two new non-ANSI-standard functions have been added:
unsigned int rand8(void);    // pseudo random number between 0..255
unsigned int rand1(void);    // pseudo random decision between 0..1
void srand(unsigned long seed); // set starting value for the sequence
```

Remark: in embedded design often the standard 'rand()' is too slow. For this we have added the 'rand8()' and 'rand1()' function. The timing on a standard 8051 is abt.: 'rand()': 500 cycles, 'rand8()': 250 cycles, 'rand1()': 35 cycles. So 'rand1()' is fast enough even for a audio-noise-generator...

Conversion functions for strings to integer or long values

```
int atoi(far char* pc);
long int atol(far char* pc);
```

The following functions are non standard, but often quite useful. 'putsl()' is like 'puts()', except it does not add the new line char at the end. The 'inputse()' can be used to retrieve a

input from the user to the string, where '*pc' points to, of size 'max'. The result is the length of the user's input (at maximum this is 'max'-1). The string has a '\0' char at the end.

```
void putsl(far char *ps); // Non Standard

unsigned char inputse(far char *pc,unsigned char max); // Non Standard
```

Remark: inputse() is used in the demo 'SRC\MINI535\M535V3.C'

A standard wait-function has been added:

```
void _wait_ms(unsigned int);
```

This function will always wait a given time in msec. This is accomplished by a macro CPU_NSEC, which holds the 'mean instruction time' for a 1-cycle instruction in nsec (for a 12 MHz generic 8051 this is 1000, which is the default too..., a 24 MHz generic 8051 would have 500). See comments for V1.10.9 too. This parameter can be found in MakeWiz's 'C-Compiler' Tab.

Please note: For some High-speed cores this value might differ, i. e. the MSC1210 requires CPU_SEC set to 620, although running with a 11.0592 MHz crystal (see SRC\MSC\ELMET*.MAK)... and additonally, the nsec value will only be accepted, if <=1000. For very slow CPUs leave nsec to its default and calculate waiting times manually (i.e. for a 2 MHZ 8051 call '_wait_ms(10)' to wait 60 msec...

Important: A problem was discovered on the ADuC's from Analog-Devives (may occure on other CPUs with a 3-byte DPTR-register too, because the 3-byte DPTR of the ADuC is not totally compatible with the "generic" 8051): '_wait_ms()' uses the DPTR-register as an up counter. On the ADuC the 16-bit overflow causes an increment of the DPP register. Because the assembler instruction "mov DPTR,#value" does not change the DPP, all further DPTR indirections (like access to internal XRAM) may fail, because DPP>0 now implies access to external XRAM...

string.h

A few ANSI standard string functions...

```
int strlen(far char* px);
int strcmp(far char* pa,far char* pb);
int memcmp(far char* pa,far char* pb, int n);
far char* strcpy(far char * pdst, far char * psrc);
```

A non ANSI byte move (note: 'src' is the first argument!). But also quite common.

```
void bmove(far void * _psrc, far void * _pdest, unsigned int count);
```

Remark: all memory move functions use generic pointer, so memory can be moved from and to everywhere. If the memory type of the source and destination is known (i.e. both external RAM), a more specific move function could run very much faster. We will supply these, in the next version of µC/51.

ctype.h

```
char tolower(char c);
char toupper(char c);
```

stdarg,h

Defines ANSI compatible access to variadic arguments

```
va_start(y,y);
va_arg(x,y);
va_end(x);
```

bin_safe.h

This file declares only one function for the 'binary safe' (as stated before):

```
unsigned char _bin_safe(void); // success: return 0
```

math.h

All math functions have a precision of at least 7 significant digits. All algorithms use industrial approved iterations. If you need a special mathematical function, let us know!

Constants:

```
M_PI_2    1.570796326794895  // ANSI
M_PI      3.14159265358979   // ANSI
M_TWO_PI  6.28318530717958   // Constant NOT ANSI
```

Functions (all according to the ANSI standard):

```
float atof(far char* pc);    // convert a string to float
float sqrt(float f);         // square root

float sin(float x);          // sin() function in radiants (standard)
float cos(float x);          // cos()
float log(float x);          // natural logarithm
float log10(float x);        // decimal logarithm
float exp(float x);          // exponential function
float pow(float x);          // power function
```

8051 specific

irq52.h

To bind an interrupt to an interrupt function, 'irq52.h' defines a macro. 'name' is the name of the function, 'loc' is either the address of the interrupts or its name (like INT0, SERIAL, ...)

```
IRQ_VECTOR(name,loc);
```

reg51.h, reg52.h, reg535.h, reg552.h, 89C51RD2.h (ATMEL)

Header files for the most common CPUs, other CPUs will follow. The header files for very specific CPUs (like the both mixed signal parts), are located in the directory above their demos. Assembler definitions with extension *.def are available too!

sys51.h

This include file is holding general 8051 system specific things, not covered by the included files above. Currently there are only 2 entries, an intrinsic 'nop' and a definition for accessing 2-bytes SFRs (such as the DPTR or TH/L pairs)

```
void _nop_(void); // will insert a 'pure' NOP-instruction
SFR16_READ(a);      // read 16 bits from sfr a
SFR16_WRITE(a,b);   // write 16 bits b to sfr a
```

kar.h

Include this header file if using sourcecodes from other compilers.

Appendix A: Migrating from other compilers

As stated before, µC/51 was designed to be as close to the ANSI standard as possible on the 8051. That is why most items in converting sourcecodes from other compilers concern non standard language extensions of these compilers. Usually the µC/51 compiler will tell you, what it does not like. Some of the most important things you should know:

Memory usage - memory models

µC/51 offers only two memory models: 'small' (this is the default) and 'large'. The only difference between the two models is the type of memory for the local variables. For 'small' it is the internal RAM, for 'large' it is external RAM.

Non-constant Global variables will always be placed in external RAM, except something else is explicitly specified. Globals might be placed in the internal RAM by the memory modifier 'near', in the indirect internal RAM with 'inear', ...

Constant global variables are located in the code memory.

Bit variables are declared as 'unsigned char bit'.

A useful header file

We have supplied a header file 'kar.h' that you may include if using sourcecodes form other compilers. It defines some of the 'market leader's' keywords:

```
#define sfr     near unsigned char    // needs an absolute address
#define sbit    unsigned char bit     // dto.
```

```
#define data    near                // internal lower RAM (128 bytes)
#define idata   inear               // indirect internal RAM (256 bytes)
#define bit     unsigned char bit
```

Absolute addresses

Assigning an absolute address to a variable is done by the '@' operator (i.e. 'near unsigned char P0 @ 0x80'). After '@' only a number, a constant expression or a defined constant (i.e. '#define SCL 0xB0+7', 'unsigned char bit scl_bit @ SCL') is allowed.

Interrupts

Interrupts might be written in µC/51. The vector is not included in the declaration, binding an interrupt to a function requires the macro 'IRQ_VECTOR' (as mentioned in the previous text).

Assembly language

µC/51's assembler uses other directives. However, mnemonics are full compatible to the 8051 standard. Best way to convert assembler files to µC/51 is to use a text editor and some bulk 'search and replace'.

Constant strings

As stated in the paragraph about memory types, there is a logical incompatibility in some other compilers, concerning constant strings. They allow the definition of a constant string as

```
code char* pc="Oops!";     // some other compilers! Attention!
```

µC/51 will accept this, and it will run as expected, nevertheless µC/51 will treat 'pc' as a 'pointer to code, located in xram'. Whereas the definition

```
code char pc[]="HELLO";  // string, located in code, 8051 recommended!
```

will declare 'pc' as a constant string, requiring no ram.

Appendix B: Distributors

Dontronics (Australia): www.dontronics.com

Grifo (Italy): www.grifo.it , www.grifo.com

HW.CZ (Czech): www.hw.cz , www.hw-server.com

Blitzlogic (Malaysia): www.blitzlogic.com

Lextronic (France): www.lextronic.fr

Batronix (Germany): www.batronix.de , www.batronix.com

8052.com's online store: www.8052.com/store

Appendix C: Revision history

V1.10

- First official release

V1.10.1, V1.10.2

- Fixed some minor bugs

V1.10.3

- Made some corrections in the documentation
- Added more demos (especially Single-Chip controllers)

V1.10.4

- Fixed some minor bugs
- Added more demos for the MSC1210 from Texas Instruments

V1.10.5

- Added some comments in several demos

V1.10.6

- Updated docu and demos for the C515 CPU (with CAN Bus), used on MINI535 V3.2

V1.10.7

- Updated docu (mainly about "constant strings")
- Fixed some minor bugs

V1.10.8

- Corrected library 'ser_iod.lib' (for serial I/O, using the SLD51 debugger): some non visible characters invoked the debugger
- UmShell: added a tool 'farg.exe' to omit some problems with the 'echo' (called from uMake), if uC/51 is installed to other drives than Windows.

V1.10.9

- Added text about updates in the documentation
- Fixed a (very rare) error concerning common subexpressions in xdata memory
- Added definitions for 'putchar()' and 'getchar()' in 'stdio.h'

- Fixed the macro name for pre-startup()-functions to 'STARTUP_FIRST' in the docu (see 'The job of startup()').
- Corrected the docu for '#pragma cpu51' directives
- Added include file 'sys51.h' for system specifics (like '_nop_()')
- The 'rand()' function has been changed to use binary polynominals, thus making it quite faster with improved spectrum. Two practical (although non ANSI-functions) have been added: 'rand8()' which gives an 8-bit uchar random value, and 'rand1()', which is very fast and gives a simple binary decision (0/1), see 'stdio.h' for details.
- Added a note about the initialisation values of register banks 2 and 3, which might be used for global variables (see: 'register usage') .
- old I2C LCD project removed and replaced by:
- new LC-Display for universal I/O (I²C, Ports, ...) LCD2
- Added documentationd and demos for the ELEKTOR-METER (see 'SRC\MSC120\ELMET.PDF)
- _wait_ms() generalised. This standard function will now wait quite precise on any 8051-core. For this the macro CPU_NSEC must be defined, as shown in many demos. This macro holds the "mean instruction time" in nsec (for a 12 MHz generic 8051 this is 1000). Soem examples for CPUs (like for a 24 MHz generic 8051 it is 500 or for a 11.0592 MHz MSC121x (high speed core) it is about 620 (not precisely), for a MSC121x with 1.8432 MHz it is about 3720. If not defined, CPU_NSEC is set to its default value of 1000.

V1.10.10

This version is mainly a service release and should be the last before the V1.20 with a new GUI will be released, which is currently already under construction...

Because some of our projects have become larger than expected, the V1.20 (GUI) and V1.30 (Flash-SLD51) have been delayed. Sorry, we will publish them A.S.A.P.

- Fixed an unsevere error in the call-graph-algorithm
- Support for C515C improved: OS515 updated for C515C with 57600 Bd. Now the demo MINI535.C can be used with this Baudrate (with some limitations, see sourcecodes).
- Some users have complained about the unused register banks 0 and 1. There is always the possibility to use the '@'-directive to place variables manually there, but one user asked for a more automatic solution. This is our tip: Enlarge DRAM if no bits variables are used:

```
#asm
     .segment nearbss, org $10, sclass dram
#endasm
```

  If this directive is places anywhere in one sourcecode, the linker will find it and not use the default address at $20. But this is only allowed if no bit variables are used!

- The NVOLA demo has been added for the MSC121x-Family. This demo shows how to use the MSC's internal Flash for storing code and data! To use this feature, a new option has been added to the BIN2HEX converter: '-s'. This option compresses HEX files, by omitting sections with $FF bytes (because this is the default value for empty Flash memory). So: if using Flash memory, speed up the download, by setting the macro 'B2HFLAGS= -s' in your makefile!

V1.10.11

This version is mainly a service release and should (really) be the last before the V1.20 with a new GUI will be released, which is currently already under construction...

- The linker reserved sometimes a few unused bytes in the internal RAM
- _wait_ms() not useable on ADuC's CPU form Analog Devices
- Added and completed include-files for ADuC 812/831/841
- added forgotten 'sys51.h' to the installation

V1.10.12

- Fixed some minor (cosmetically) bugs
- Added header files for 80C552, 89C51RD2 (ATMEL) in '\include'
- Added the ELMET485 demo for the ELEKTOR board's RS485
- Added licence transfer (see "Transferring a license")

V1.10.13

- Added a short note in 'Interrupts in C' about IRQs at startup. Might be important to find mysterious bugs...

V1.10.15 (V1.10.14 omitted)

- First time the new TCP/IP Stack was included (although without docu)
- WebCode HTML->OBJ converter added
- JFE Editor added

V1.20.20 (V1.10.16-19 omitted)

- Added the MakeWiz
- ELM_FLEX -Demo added (MSC1210 based Webserver) with 16kB demo code extension
- Demos for the uPSD-Family from ST added
- Header Files for the ADuC family moved to "include"
- Header Files for the MSC121x family moved to "include"
- Sourcecode for the Flash_Toolbox added (Flash_TB.S51)
- Header files for ATMEL's AT89C51xD2 added

V1.20.01

- Added command line support for FlashMon (parameters FILENAME, -cXX, -bXX)
- Add docu for the FlexGate TCP/IP Stack
- Added links to all docs in the start menue

V1.20.02

- Corrected the „large memory model switch“ bug in MakeWiz
- Minor changes
- Headers for AT89C51RDs (Atmel) (*.h)

- Headers for AT89C51CC03 (Atmel) (*.h and *.def)

V1.20.03

- Added OMF51-Support for SiLabs-IDE
- Added header files for SiLabs C8051Fxxx

V1.20.04

- Added source codes and doku for the FlexGate III ('...src\ed2\flexgateIII.pdf')

Still to do or under current development:
- Doku TCP/IP-Stack for 100/10 Mbit devices
- Complete TCP/IP-Stack docu for newly added UDP/Phy-Layer Support
- Add more demos and doku for SiLabs

Appendix D: A demo of µC/51's optimiser

In this appendix, we made a comparison with the KXXX compiler V6.21. You can trace the results, they offer a 2kB limited demo version...

The test function is some kind of "bit banger": shift something in and out...

This is the result of KXXX V6.21 - it needs 35 bytes to solve the problem:

```
near unsigned char bit  out_bit ;
near unsigned char bit  in_bit ;

// Simple 8-Bit-Banger
unsigned char test(unsigned char ob){
   int i;
   unsigned char ib;
   for(i=0;i<8;i++){
           out_bit=(ob&128);
           ib<<=1;
           if(in_bit) ib|=1;
           ob<<=1;
   }
   return ib;
}
C51 COMPILER V6.21  BANG


ASSEMBLY LISTING OF GENERATED OBJECT CODE

            ; FUNCTION _bang (BEGIN)
;---- Variable 'ib' assigned to Register 'R6' ----
;---- Variable 'ob' assigned to Register 'R7' ----
;---- Variable 'i' assigned to Register 'R2/R3' ----
0000 E4                 CLR     A
0001 FB                 MOV     R3,A
0002 FA                 MOV     R2,A
0003          ?C0001:
0003 EF                 MOV     A,R7
0004 33                 RLC     A
0005 9200         R     MOV     out_bit,C
0007 EE                 MOV     A,R6
0008 25E0               ADD     A,ACC
000A FE                 MOV     R6,A
```

```
000B 300003      R     JNB     in_bit,?C0004
000E 430601            ORL     AR6,#01H
0011           ?C0004:
0011 EF                MOV     A,R7
0012 25E0              ADD     A,ACC
0014 FF                MOV     R7,A
0015 0B                INC     R3
0016 BB0001            CJNE    R3,#00H,?C0009
0019 0A                INC     R2
001A           ?C0009:
001A EB                MOV     A,R3
001B 6408              XRL     A,#08H
001D 4A                ORL     A,R2
001E 70E3              JNZ     ?C0001
0020           ?C0002:
0020 AF06              MOV     R7,AR6
0022           ?C0005:
0022 22                RET
               ; FUNCTION _bang (END)
```

And this is the result of µC/51 V1.10 - we need only 25 bytes ;-):

```
                         :>#define _P1_B6 0x86
                         :>#define _P1_B7 0x87
                         :>bit unsigned char out_bit @ _P1_B6;
                         :>bit unsigned char in_bit @ _P1_B7;
                         :>

                    ...

                         :  .segment __bang
                         : _bang: ; (leaf function) unsigned char bang(unsigned
char)
                         :  ; parameter 'ob' in 'R7' assigned to 'R5'
co:8007: ad 07           :  mov R5,AR7
                         :  ; variable 'ib' assigned to register 'R1'
                         :>
                         :>// Simple 8-Bit-Banger
                         :>unsigned char bang(unsigned char ob){
co:8009: 7b 08           :  mov R3,#8
                         : ?3:
                         :>        int i;
                         :>        unsigned char ib;
                         :>        for(i=0;i<8;i++){
co:800b: ed              :  mov A,R5
co:800c: 33              :  rlc A
co:800d: 92 86           :  mov 134,C
                         :>                out_bit=(ob&128);
                         :>                ib<<=1;
co:800f: e9              :  mov A,R1
co:8010: 29              :  add A,R1
co:8011: f9              :  mov R1,A
                         :>                if(in_bit) ib|=1;
co:8012: 30 87 03        :  jnb 135,?7
                         :
co:8015: 43 01 01        :  orl AR1,#1
                         : ?7:
                         :>                ob<<=1;
co:8018: ed              :  mov A,R5
co:8019: 2d              :  add A,R5
co:801a: fd              :  mov R5,A
                         :>        }
co:801b: db ee           :  djnz R3,?3
                         :>        return ib;
co:801d: af 01           :  mov R7,AR1
co:801f: 22              :  ret
                         : ; end of function bang
                         : ; used: R-1-3-5-7 BR-------- ACC PSW
```

µC/51's optimisers are still not fully implemented (some parts are still completely missing). The compiler is yet a learning child. Many of the optimisers are founded on some kind of expert system. As you might see, the system is working, but the collections of rules, these

expert systems are based on, are still quite small and in other cases, the results will not be as much as 40% better. But µC/51 has the ability to do it and: we have first V1.xx yet...

END